Design and application of convergent cellular automata

From morphogenesis to the design of reliable electronic systems and self-assembling robotics

David Huw Jones

A thesis presented for the degree of

Doctor of Philosophy

Centre for Electronic Systems

University of Durham

England

February 2009

Dedicated to

My family, my friends, my colleagues. Everybody who made this possible.

Design and application of convergent cellular automata

From morphogenesis to the design of reliable electronic systems and self-assembling robotics

David Huw Jones

A thesis presented for the degree of
Doctor of Philosophy

Abstract

Systems made of many interacting elements may display unanticipated emergent properties. A system for which the desired properties are the same as those which emerge will be inherently robust. Currently available techniques for designing emergent properties are prohibitively costly for all but the simplest systems.

The self-assembly of biological cells into tissues and ultimately organisms is an example of a natural dynamic distributed system of which the primary emergent behaviour is a fully operational being. The distributed process that co-ordinates this self-assembly is morphogenesis. By analysing morphogenesis with a cellular automata model we deduce a means by which this self-organisation might be achieved.

This mechanism is then adapted to the design of self-organising patterns, reliable electronic systems and self-assembling systems. The limitations of the design algorithm are analysed, as is a means to overcome them. The cost of this algorithm is discussed and finally demonstrated with the design of a reliable arithmetic logic unit and a self-assembling, self-repairing and metamorphosising robot made of 12,000 cells.

Declaration

The work in this report is based on research carried out at the University of Durham, the School of Engineering, the Centre for Electronic Systems, England. No part of this report has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

List of publications

Prior publications of the author and supervisors (David Huw Jones, Richard McWilliam and Alan Purvis) are listed below:

1. **Journal paper**: Design of convergent cellular automata, *Biosystems*, December 2008.
2. **Invited chapter**: Design of self-assembling, repairing and metamorphosising Arithmetic Logic Unit, *InTech publishing* (expected publication July 2009).
3. **Patent (pending)**: Self-repairing electronic data systems. Patent GB0802245.1
4. **Conference paper**: A bio-inspired model for the design of self-assembling systems, *Proceedings of BioNano 2009*.
5. **Conference paper**: Mimicking the community effect, *Proceedings of BioSys-Bio 2009*.
6. **Conference paper**: Mimicking morphogenesis for the design of robust electronic circuits, *World academy of science, engineering and technology*, July 2007.

Work by the author has been cited in:

1. Dependability analysis of a safety critical system: the LHC beam dumping system at CERN: http://cdsweb.cern.ch/record/997680/files/thesis-2006-054.pdf

Acknowledgements

The list is endless. To name but a few... Professor Alan Purvis, Doctor Richard McWilliam, the Jones clan, Rachael Stephenson, Chris Sparks, Ruth Vonberg, Paul Harford, Katie Rutlidge, Dr Vernon Armitage, Dr James Blowey, Dr Dirk Schutz, Joe Milbourn, Richard Curry, Christopher Paterson.

Contents

List of Figures

Chapter 1

Introduction and objectives

> The core idea is that nature, imaginative by necessity, has already solved many of the problems we are grappling with. Animals, plants, and microbes are the consummate engineers. They have found what works, what is appropriate, and most important, what lasts here on Earth. This is the real news of biomimicry: After 3.8 billion years of research and development, failures are fossils, and what surrounds us is the secret to survival. [Ben98]

Engineering is a top-down approach to systems design. Given a specification, the problem is first broken down into a number of smaller, solvable, problems; their solutions are then re-assembled to create a complete design. Usually the sub-units cannot be evaluated against the complete task specification and the failure of any one can cause the entire system to fail. In contrast, biology uses a bottom-up approach: starting with fields of homogeneous cells, co-ordinated co-operation produces structures that achieve solutions to complex specifications. Able to grow, renew and self-repair, biological systems can create large, extremely reliable and complex, systems. The mechanism that makes this reliability possible is morphogenesis.

1.1 Morphogenesis

Morphogenesis is a distributed chemical process that is responsible for co-ordinating the self-assembly and self-repair of biological systems. The abbreviated injury score (AIS) [MSP+89] can be used to quantify just how reliable self-repairing morphogenetic self-repairing systems (such as the human liver) are. It is a measure of the severity of an injury used as a tool for triage in accident and emergency departments. Table 1.1 is an example AIS, the criteria for assessing the damage to a human liver. Results from clinical trials compiled by the Association for Automotive Medicine (AAAM) show an approximate correlation (see figure 1.2) between the AIS of an injury and the probability of surviving it.

Grade	Type of injury	Description of injury
1	Laceration	Tear less than 1cm deep
2	Laceration	Tear 1 - 3cm deep, less than 10cm in length
3	Laceration	Damage of 25% to 75% of the hepatic lobe
4	Laceration	Damage to more than 75% of the hepatic lobe
5	Laceration	Damage to the central major hepatic veins
6	Vascular	Split into two or more large pieces

Figure 1.1: Abbreviated Injury Score (AIS) for liver damage

Another demonstration of the reliability of morphogenesis is evident in the salamander family. Salamanders can regrow the same limbs repeatedly, as well as their tail, jaw, and the lenses and retinas of their eyes. In terms of body mass alone, the salamander can regenerate approximately 60% of itself in the event that it is damaged.

A final, remarkable, demonstration is the self-repair capability of the ascidian (a type of marine filter feeder). They have been reported to regenerate from just partial blood cells to give rise to a fully functional organism [BC36].

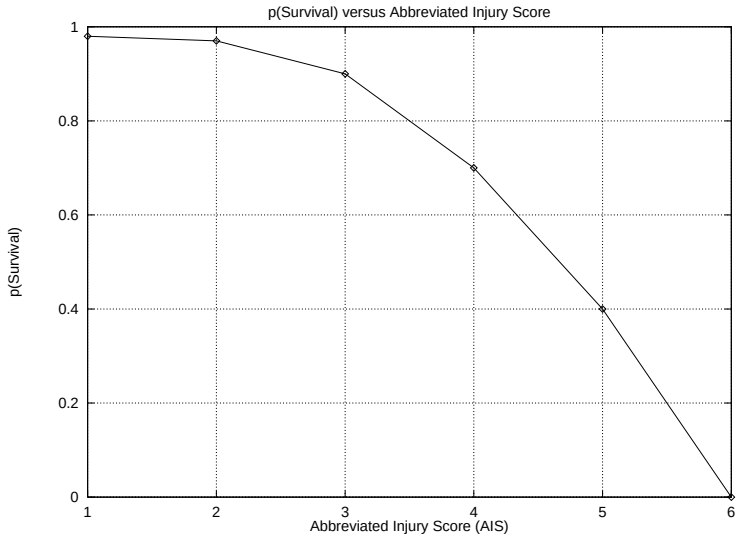


Figure 1.2: p(Survival) v. Injury score. Data based on clinical trials assembled by AAAM [SWMK05].

1.2 Biomimicry

Biomimicry is the study and imitation of natural systems. This is not a new idea: Leonardo da Vinci studied the flight of birds before designing the Ornithopter [Cut56], figure 1.3. More recent studies include the design of artificial neural networks and evolutionary computation. Another is the study of developmental biology, of which morphogenesis is a part.

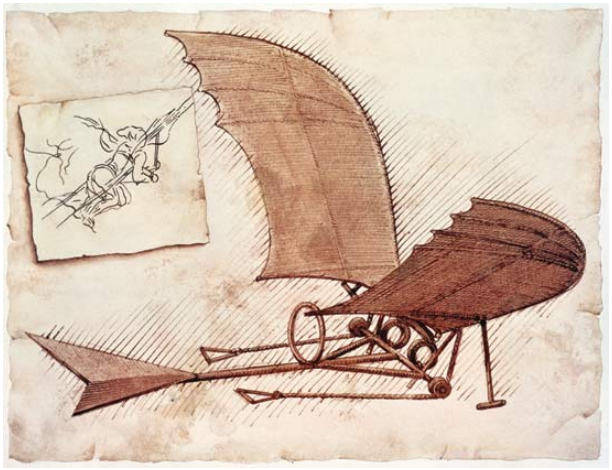


Figure 1.3: Leonardo da Vinci's bio-inspired plane

This thesis is an exercise in biomimicry: the goal is to be able to mimic morpho-

genesis in electronic systems, so as to improve their reliability. The first stage of biomimicry is a study of the subject and the development of an appropriate model.

Various models already exist for morphogenesis. Alan Turing [Tur50a] was the first to propose a model for morphogenesis that consisted of multiple point sources of chemicals that diffuse radially and interact with each other. This study, and subsequent work by Gurdon [HG90] and Meinhardt [Mei82], is the subject of the second chapter of this thesis.

More recent models have been computational. Researchers in this field have used supervised evolutionary algorithms to evolve local rules against a cost function that is defined as the difference between the desired global arrangement and the global arrangement that results from the systems development. Their models, algorithms and results are the subject of the third chapter of this thesis.

1.3 Objectives of research

If the mainstream bio-inspired techniques are neural networks and evolutionary computation, developmental biology is one of the more obscure. This is principally because the relationship between the local rules obeyed by each cell and the resulting global arrangement is unknown. The lack of a suitable mapping from local rules to global arrangement has cost the field in both the possible size and complexity of the system, and also in the computational time necessary to search the possible local rules for a solution. Thus one goal of the research presented here is to replace evolutionary algorithms with a deterministic approach to designing morphogenesis-inspired systems; this is achieved with a cellular automata model.

The invention of John von Neumann, cellular automata (CA), were used to study self-replication. CA typically consist of arrays of identical computing cells that locally and synchronously interact at discrete time intervals to determine their state. Nowadays CA are the bailiwick of mathematicians studying chaos. Certain CA

respond to small changes in their initial conditions with a disproportionately large change to their final conditions. To the contrary, if CA are to be used as a modelling framework for morphogenesis, they will need to be insensitive to initial conditions: that is, regardless of their starting conditions, the CA must converge to the same final form. An algebraic and functional analysis of this requirement is presented in chapter four.

However, it is not enough for a CA model to simply converge to the same fixed point. To be of use, the CA must be designed to converge to a fixed point we can choose. Chapters five and six present an algebraic and deterministic design algorithm for this purpose. Chapter seven demonstrates the results with the design of various converging CA. Chapter eight demonstrates an improvement to the model that is also bio-inspired.

In order to apply this model to the design of reliable electronic systems it is necessary to infer electronic function to the form of this biological analogue. Thus, the biological tissue becomes a massive parallel arrangement of small computing units; the differentiated, fully-formed cell becomes instead an operational component of a larger system.

Chapter nine introduces existing tools for system analysis of, and design for, reliability. Chapter ten describes, demonstrates and analyses the design of a self-assembling self-repairing arithmetic logic unit (ALU) on a field-programmable gate-array (FPGA) hardware and an alternative application-specific integrated-circuit (ASIC). The design process requires a top-down discretisation of the design into constituent components and the development of a bottom-up self-assembly algorithm.

Self-assembling, self-repairing robotics has largely been the bailiwick of science-fiction writers. The design of such a system with greater than a thousand identical components is the subject of an IEEE robotics grand challenge for this century. Chapter eleven describes how the rectangular CA model of morphogenesis can be

adapted to the design of irregular three-dimensional shapes. This is then demonstrated with the self-assembly and self-repair of a model of a robot shape formed from 55,000 identical cells. Finally, this algorithm is adapted to converge to one of many forms according to the initial conditions of the first cell. This makes possible the design of metamorphosising systems: systems capable of transforming from one form to another according to the demands of the environment or the challenges of the latest task. This is demonstrated with the design of a 12,000 cell system that can transform from a robot shape into a car shape and back again.

Chapter 2

Models of Morphogenesis

> For nature, as we are so often reminded, is under no obligation to make things simple just for our convenience [Cho01].

2.1 Overview

Here we present Turing's ordinary differential equation model of morphogenesis to show that spatiotemporal patterns can be formed by simple diffusion models. This model is then elaborated upon to include Meinhardt's system of five diffusing morphogens which can better imitate observed biological patterns.

2.2 Cannibals and Missionaries

Imagine an island of cannibals and missionaries. The cannibals can have sex, creating other cannibals in the process. The missionaries cannot have sex but own bicycles; two missionaries convert a cannibal into another missionary. This is the analogy Alan Turing [Tur50a] used to describe his "Reaction-Diffusion" model of morphogenesis. On such an island pockets of cannibals surrounded by missionaries

are formed.

Listing 2.1 is the pseudo-code of an agent-based "cannibal-missionary" island model, Appendix A.1 includes a complete code listing and figure 2.1 shows an example result of the model.

```
1 Create 500 missionaries and 500 cannibals.
2 Create a world of 128x128 discrete 2D square cells.
3 Place each agent in a cell chosen at random.
4
5 For each iteration of the simulation:
6         For each agent:
7                 Move the agent left,right,up,down or leave it where it is.  Each
                      movement is equally probable, and is determined by a random number.
8         For each cell:
9                 Test for the presence of any combination of two cannibals or two
                      missionaries and a cannibal.  The agents should be tested in order
                      of age, and the combinations only apply to agents adgacent to each
                      other in this sequence.
10                For each combination of two cannibals create another cannibal in the
                      same cell.
11                For each combination of one cannibal and two misisonaries, convert the
                      cannibal into a missionary.
12
13 Output a 128x128 bit image of the world.  The cells dominated by cannibals, colour red.
        The cells dominated by missionaries, colour green.
```

Listing 2.1: Cannibals and Missionaries pseudo-code

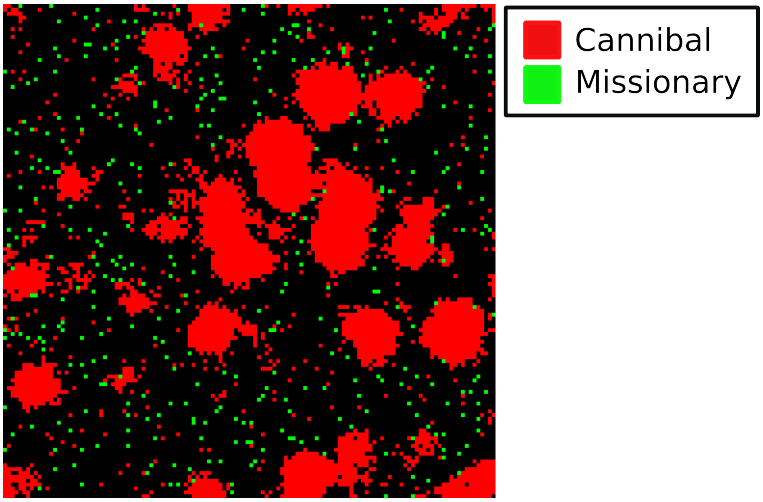


Figure 2.1: Clumping of cannibals and missionaries formed by Turing's reaction-diffusion analogy

The spotty pattern formed by the missionaries (green) and cannibals (red) is analogous to the spotty pattern observed on the skin of a cheetah.

2.3 A model of morphogenesis

Turing proposed that the spots of a cheetah, the stripes of a zebra and every other arrangement of cells within living systems could be explained by the diffusion of chemicals he named "morphogens" and their interaction with each other. He sought to show this by proving that a set of identical cells could, if isolated, have stable morphogen concentrations. However if the morphogens were permitted to diffuse between cells, the morphogen concentrations would become unstable and patterns would form.

To demonstrate this model, let us consider a simple two-morphogen (x_1, x_2) system of cells in one-dimension [1].

2.3.1 The resting state

If, for a moment, we consider small perturbations of the concentrations of the morphogens about $\bar{x_e}$, the system's resting state, we can use linear ordinary differential equations (ODEs) (2.1) to describe their reaction and diffusion.

$$\frac{d\bar{x}}{dt} = \mathbf{A}\bar{x} \tag{2.1}$$

Where $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$.

Let λ be an eigenvalue of $\mathbf{A}$. According to Cramer's rule, the system will only have non-trivial solutions if $det(\mathbf{A} - \lambda I) = 0$.

[1] This proof was derived from Turing's paper [Tur50a], the lecture notes of Dr James Blowey[Blo07] and the lectures notes of Professor Stephen Childress[Chi05]

$$det(\mathbf{A} - \lambda I) = a_{11}a_{22} - \lambda(a_{11} + a_{22}) + \lambda^2 - a_{12}a_{21} \tag{2.2}$$

Let

$$T = a_{11} + a_{22} \tag{2.3}$$

$$D = a_{11}a_{22} - a_{12}a_{21} \tag{2.4}$$

Therefore $det(\mathbf{A} - \lambda I) = \lambda^2 - T\lambda + D$ and its roots are given by:

$$\lambda = \frac{T \pm \sqrt{T^2 - 4D}}{2} \tag{2.5}$$

From (2.1) we know $x \propto e^{\lambda t}$, thus for the resting state to be stable (change from $\bar{x}_e$ is opposed), both roots of (2.2) must be negative. Therefore $D > 0$ and $T < 0$.

2.3.2 The transient state

Now let us consider larger permutations of the concentrations of the morphogens, as a result of diffusion of the morphogens between neighbouring cells.

Let each cell be a small cube of side $\triangle$ with the chemicals $\bar{x}$ distributed equally within it. Each cell has an index i from 0 to N and the cells are arranged in a 1-D loop such that the neighbours of cell N are $N - 1$ and 0.

Ficke's law states:

> The flow of chemical from one cell to another is proportional to the difference in the chemical concentrations of the two cells, the flow being from the higher to the lower.

The flux, f_j, of morphogen j into cell i, will be:

$$\begin{aligned} \text{flux } f_1 &= k_1 \triangle^2 (x_1^{i-1} - x_1^i) + k_1 \triangle^2 (x_1^{i+1} - x_1^i) && (2.6)\\ &= k_1 \triangle^2 (x_1^{i-1} - 2x_1^i + x_1^{i+1}) && (2.7)\\ f_2 &= k_2 \triangle^2 (x_2^{i-1} - 2x_2^i + x_2^{i+1}) && (2.8) \end{aligned}$$

where $\{k_1, k_2\} > 0$ are constants of proportionality. Now

$$\frac{d\bar{x}^i}{dt} = f(\bar{x}^i) + \mathbf{M}(\bar{x}^{i-1} - 2\bar{x}^i + \bar{x}^{i+1}) \tag{2.9}$$

where $\mathbf{M} = \begin{bmatrix} \triangle^2 k_1 & 0 \\ 0 & \triangle^2 k_2 \end{bmatrix}$ and $f(\bar{x}^i)$ determines the proportion of $\bar{x}^i$ that remains in cell i.

Note that because the cells form a loop, the system is closed and $\sum_i x^i$ is constant.

If the width, $\triangle$, of each cube tends to 0 we can consider the system continuous. If $s = \triangle .i$ then the flux entering the position s, $f(s)$ is given by:

$$\frac{d\bar{x}(s)}{dt} = f(s) = \mathbf{M}(\bar{x}(s- \triangle) - 2\bar{x}(s) + \bar{x}(s+ \triangle)) \tag{2.10}$$

If we expand each term into its Taylor series through terms in $\triangle^2$ then:

$$\begin{aligned} f(s) &= \mathbf{M}\left[\bar{x}(s) + \frac{d\bar{x}(s)}{ds} \triangle + \frac{d^2\bar{x}(s)}{ds^2}\frac{\triangle^2}{2} + \cdots\right] && (2.11)\\ &\simeq \mathbf{M} \triangle^2 \frac{d^2\bar{x}(s)}{ds^2} && (2.12) \end{aligned}$$

Let us assume $\triangle \to 0$, the number of cells, $N \to \infty$, $N \triangle \to L$, the circumference of the ring. We want to study the linear stability of the resulting continuous partial differential equation (PDE) in time.

$$\frac{\partial \bar{x}(t,s)}{\partial t} = \mathbf{A}\bar{x}(t,s) + \mathbf{M}\frac{\partial^2 x}{\partial s^2}(t,s) \tag{2.13}$$

Where $\mathbf{M}$ has been redefined as $\begin{bmatrix} \mu_1 & 0 \\ 0 & \mu_2 \end{bmatrix}$, $\mu_i = k_i \frac{\triangle^4}{2}$. μ_1, μ_2, the diffusion coefficients are finite and positive.

As the cells are in a loop we can look for pattern waves of the form:

$$x = e^j(\lambda t + ks)x_0 \tag{2.14}$$

Where λ is the angular frequency and k is the wave vector. If for some μ_1, μ_2, k there exists a solution to (2.13) in the form of (2.14) such that $\Re(\lambda)$ is positive, the concentrations of the morphogens within the tissue will form unstable patterns.

Substituting $\frac{d^2\bar{x}}{ds^2}$ of (2.14) into (2.13) gives:

$$\frac{d\bar{x}}{dt} = \mathbf{A}\bar{x} + M\begin{bmatrix} -k^2 & 0 \\ 0 & -k^2 \end{bmatrix}\bar{x} \tag{2.15}$$

$$= \left(\mathbf{A} - \begin{bmatrix} \mu_1 k^2 & 0 \\ 0 & \mu_2 k^2 \end{bmatrix}\right)\bar{x} \tag{2.16}$$

Again using Cramer's rule:

$$det\left(\begin{bmatrix} a_{11} - \mu_1 k^2 - \lambda & a_{12} \\ a_{21} & a_{22} - \mu_2 k^2 - \lambda \end{bmatrix}\right) = 0 \tag{2.17}$$

Which is a binomial of the eigenvalues of (2.16), λ:

$$\lambda^2 - \tau\lambda + \psi = 0 \tag{2.18}$$

where

$$\tau = (a_{11} + a_{22}) - k^2(\mu_1 + \mu_2) \tag{2.19}$$

$$= T - k^2(\mu_1 + \mu_2) \tag{2.20}$$

$$\psi = D - \mu_1 k^2 a_{22} - \mu_2 k^2 a_{11} + \mu_1 \mu_2 k^4 \tag{2.21}$$

In order for patterns to form, (2.13) must be unstable. Therefore at least one of the roots of (2.18) must have a postitive real component. As $T < 0$, $\tau < 0$. Thus we need $\psi < 0$. Therefore at least one of a_{11} and a_{22} must be positive. Because $T < 0$ at least one of a_{11} and a_{22} must be negative. Therefore $a_{11}a_{22} < 0$. Again because $\psi < 0$:

$$a_{22}\mu_1 + a_{11}\mu_2 > 0 \tag{2.22}$$

If $a_{11} < 0$ its corresponding entry in the matrix $\mathbf{B}$, $a_{11} - \mu_1 k^2 + \sigma$ is also negative. Thus the chemical, x_1 will decay to the rest state $\bar{x_e}$ in the absence of x_2. Turing called x_1 the inhibitor chemical and x_2 the activator chemical.

2.3.3 Turing instability inequalities

From (2.22) and (2.3) one can deduce that, for the system to form stable patterns $\mu_2 < \mu_1$. Thus the inhibitor chemical must diffuse faster than the activator chemical. Referring back to the "cannibals and missionaries" analogy, the presence of two missionaries in any given square inhibits the progress of the diffusion of the cannibals across the island, however because the inhibitor-missionaries own bicycles they diffuse more rapidly than the activator-cannibals.

Equation (2.21) is a quadratic in k^2, if we differentiate it with respect to k^2 and set it equal to zero, the minima can be found:

$$\frac{d\psi}{dk^2} = 2\mu_1\mu_2k^2 - (\mu_1 a_{22} + \mu_2 a_{11}) = 0 \tag{2.23}$$

Thus the minimum value of ψ is:

$$\psi_{min} = D - \frac{(a_{22}\mu_1 + a_{11}\mu_2)^2}{4\mu_1\mu_2} \tag{2.24}$$

As ψ must be less than zero:

$$a_{22}\mu_1 + a_{11}\mu_2 > 2\sqrt{\mu_1\mu_2 D} \tag{2.25}$$

The Turing inequalities for a two morphogen system to show diffusion instability are thus:

$$a_{11} + a_{22} < 0 \tag{2.26}$$

$$a_{11}\mu_2 + a_{22}\mu_1 > 0 \tag{2.27}$$

$$a_{11}a_{22} - a_{12}a_{21} > 0 \tag{2.28}$$

$$a_{22}\mu_1 + a_{11}\mu_2 > 2\sqrt{\mu_1\mu_2(a_{11}a_{22} - a_{12}a_{21})} \tag{2.29}$$

2.3.4 An example reaction-diffusion system

An example two-morphogen system that meets these instability inequalities has the following reaction equations:

$$\frac{dx_1}{dt} = (\frac{1}{2} - x_1 + x_1^2x_2) \tag{2.30}$$

$$\frac{dx_2}{dt} = (1 - x_1^2x_2) \tag{2.31}$$

$$\tag{2.32}$$

To find the unique equilibrium we set (2.30) and (2.31) to zero. Thus $x_{e1} = \frac{3}{2}$ and $x_{e2} = \frac{4}{9}$.

Assuming the system is linear about $\bar{x_e}$ we can differentiate (2.30) and (2.31) to form the jacobian matrix, **A**.

$$\mathbf{A} = \begin{bmatrix} \frac{\partial dx_1}{dx_1} & \frac{\partial dx_1}{dx_2} \\ \frac{\partial dx_2}{dx_1} & \frac{\partial dx_2}{dx_2} \end{bmatrix} = \begin{bmatrix} -1 + 2x_1x_2 & x_1^2 \\ -2x_1x_2 & -x_1^2 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{9}{4} \\ \frac{-4}{3} & -\frac{9}{4} \end{bmatrix} \tag{2.33}$$

This meets the inequalities:

$$a_{11} + a_{22} = -\frac{23}{12} < 0 \tag{2.34}$$

$$a_{11}a_{22} - a_{12}a_{21} = \frac{27}{12} > 0 \tag{2.35}$$

If $u_1 = 1$, we can calculate u_2 from (2.29).

$$\frac{-9}{4} + \frac{u_2}{3} > 2\sqrt{u_2\frac{27}{12}} \tag{2.36}$$

This is true for all $u_2 > \frac{27}{4}(7 + 4\sqrt{3})$, a value which also meets the final inequality:

$$a_{11}\mu_2 + a_{22}\mu_1 < \frac{27}{12}(7 + 4\sqrt{3}) - \frac{9}{4} > 0 \tag{2.37}$$

Listing 2.2 is the pseudo-code of a cell-based "reaction-diffusion" model of this example, Appendix A.1 includes a complete code listing, figure 2.2 is the results of the model.

```
1  Create a world of 64x64 discrete 2-d square cells.
2  Set the morphogen values a,b, in each cell to be 4.
3  Assign each cell a diffusion constant, B = 12 +- 0.2.
4  Other diffusion constants are: Da, Db, s.
5
6  For each iteration of the simulation:
7          For each cell:
8                  aSum = sum(a values of neighbouring cells (North, South, East, West))
9                  bSum = sum(b values of neighbouring cells (North, South, East, West))
10                 a = a - s*(0.5-a+a*2*b)+Da*(aSum - 4*a)
11                 b = b - s*(1-a**2*b)+Db*(bSum - 4*b)
```

```
12
13 Output a 64x64 bit grey-scale image of the world that corresponds to the strength of
       morphogen b in each cell.
```

Listing 2.2: Reaction-Diffusion example pseudo-code

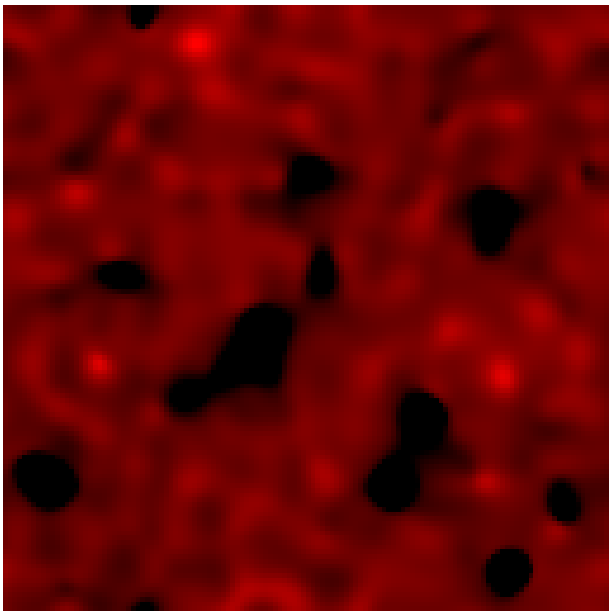


Figure 2.2: Example of Turing's reaction-diffusion equations

2.4 Experimental evidence for morphogenesis

A two-morphogen system exists in a developing human body, shortly after the creation of the blastula. Two proteins, the BCD protein and the HB-M protein, create a localised determinant centered about the zygote [Gri76]. Since these proteins have different diffusion rates, and interact with each other, spatial concentration patterns form. Different concentrations activate different genes in the genome of each cell. These gene selections in turn correspond to different types of cell. Thus the beginnings of form and cellular differentiation appear in the developing embryo.

A three-morphogen system exists in the developing fruit fly. The morphogens, bicoid, eve and caudal, are important for patterning the head, thorax and abdominal regions of the embryo [RHT+97]. Figure 2.3 shows the concentrations of each mor-

phogen shortly after fertilisation. Figure 2.4 shows various other morphogenesis patterns that have been sampled during the development of the fruit fly.

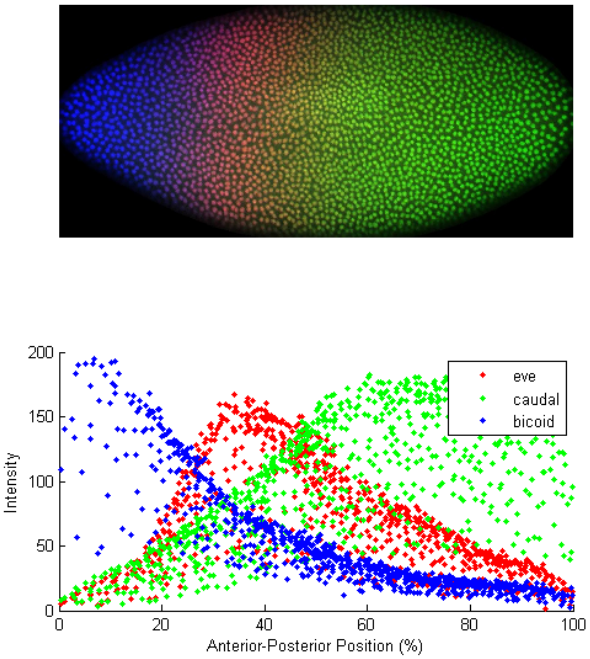


Figure 2.3: Anterior-posterior determination in fruit flies

Image courtesy of Dr. Eric Lecuyer, Canadian Institute of Health Research

2.5 A more comprehensive model of the developmental cycle

But experimental results suggests that control of the developmental cycle is not as simple as systems of interacting morphogens. Most tissues have asymmetric distributions of various properties: for example, in a hydra the nerve cell density is much greater in the head than the body. Transplantation experiments [Mor04] suggest that this polarity affects the decision of where to form structures relative to the tissue.

Further work by the experimental biologist John Gurdon [Gur68] suggests that the concentration levels of morphogens within a tissue are typically very small and

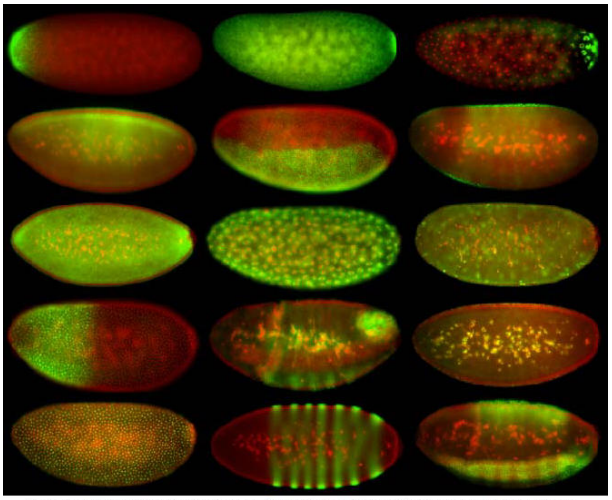


Figure 2.4: Morphogenesis patterns in fruit flies

Image courtesy of Dr. Eric Lecuyer, Canadian Institute of Health Research

undetectable by the majority of cells. To ensure each cell in this tissue develops in line with the rest of its surrounding cells, each cell that detects the presence of a signalling protein transmits a chemical signal to its neighbouring cells. This is known as cell-to-cell communication, or the community effect.

Meinhardt [Mei82] sought to incorporate these theories of development into a more comprehensive model than that proposed by Turing. The following model, one of many he proposed, uses five reaction-diffusion equations to form stripy patterns.

$$\frac{\partial g_1}{\partial t} = \frac{cg_1^2}{rs_1} - \alpha g_1 + D_g \frac{\partial^2 g_1}{\partial x^2} + \rho_0 \tag{2.38}$$

$$\frac{\partial g_2}{\partial t} = \frac{cg_2^2}{rs_2} - \alpha g_2 + D_g \frac{\partial^2 g_2}{\partial x^2} + \rho_0 \tag{2.39}$$

$$\frac{\partial r}{\partial t} = \frac{cg_1^2}{s_1} + \frac{cg_2^2}{s_2} \tag{2.40}$$

$$\frac{\partial s_1}{\partial t} = \gamma \frac{g_1 - s_1}{D_s} \frac{\partial^2 s_1}{\partial x^2} + \rho_l \tag{2.41}$$

$$\frac{\partial s_2}{\partial t} = \gamma \frac{g_2 - s_2}{D_s} \frac{\partial^2 s_2}{\partial x^2} + \rho_l \tag{2.42}$$

g_1 and g_2 are short-range activator substances. They form mutually exclusive feed-

back loops. Each is subject to long-range inhibitor substances, s_1 and s_2. That they are mutually exclusive is ensured by equation (2.40).

Listing 2.3 is the pseudo-code of a cell-based five-morphogen "reaction-diffusion" model described above, Appendix A.1 includes a complete code listing, figure 2.5 shows a result of the model with different diffusion rates D_g and D_s.

```
1 Create a world of 128x128 discrete 2-d square cells.
2 Set the morphogen values g1,g2,r,s1,s2 in each cell to be 4(+/- 0.5).
3 Diffusion constants are: c,alpha,Dg,rho0,beta,gamma,Ds,rho1
4
5 For each iteration of the simulation:
6         Diffuse g1,g2,r,s1,s2 across the world
7
8         For each row:
9                 calculate the second derivatives of g1,g2,r,s1 and s2 in the x axis.
10
11        For each cell:
12                g1 += c*s2*g1*g1/r - alpha*g1 + Dg*d2g1 + rho0
13                g2 += c*s1*g2*g2/r - alpha*g2 + Dg*d2g2 + rho0
14                r  += c*s2*g1*g1 + c*s1*g2*g2
15                s1 += gamma*(g1-s1)*Ds*d2s1 + rho1
16                s2 += gamma*(g2-s2)*Ds*d2s2 + rho1
17
18 Output a 128x128 bit black and white image of the world.  If the strength of either g1
       or g2 in each cell is greater than that of the average for the world, the cell-
       pixel is black.  Otherwise the cell-pixel is white.
```

Listing 2.3: Reaction-Diffusion example pseudo-code

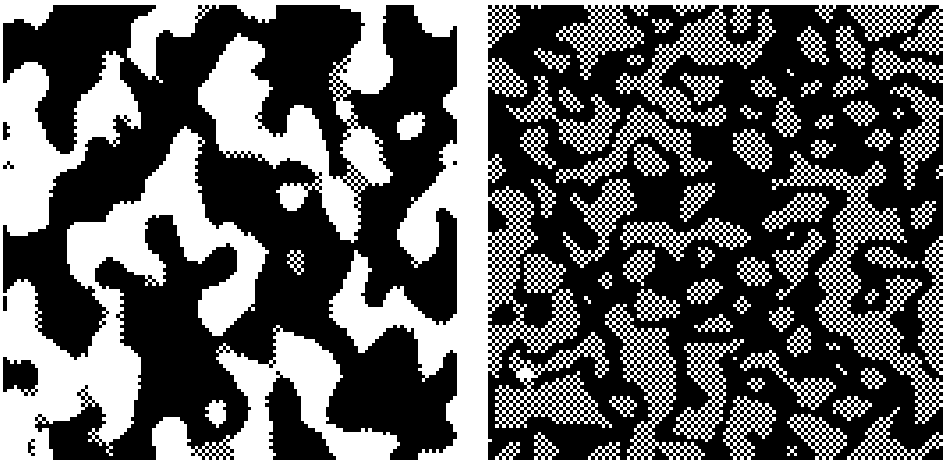


Figure 2.5: Example of Meinhardt stripes

2.6 Conclusions

In 1952 Turing published the paper "The Chemical Basis of Morphogenesis" that described a mathematical model of how morphogenesis *might* work. Despite being largely condemned by biologists at the time his ideas, though incomplete, have been shown to be largely correct. Further work by biologists such as Gurdon and mathematicians such as Meinhardt have developed Turing's ideas to better match the experimental evidence.

Chapter 3

Design and simulation of morphogenesis

Whereas Turing's investigation of morphogenesis relied on mathematics with only a cursory use of the computer, modern investigations have used computers to a much greater extent. This is perhaps because of the greater available computational power — Turing only had access to the "Manchester Automatic Digital Machine", a 40-bit 800Hz computer composed of 4,200 vacuum tubes, on which to demonstrate his model. This chapter presents a summary of modern attempts to model morphogenesis using computer simulations.

3.1 A morphogenesis simulation framework and results

The influences on a developing biological cell can be grouped into the following categories:

1. **Genetic**. A pluripotent stem cell can differentiate into a diverse range of

specialised cell types. In contrast, a cell formed by mitosis can only take the form of its parent cell.

2. **Mechanical**. Certain cells can move about a tissue during development, thus affecting orientation and location within asymmetric tissues and choice of neighbouring cells. Odell [OOAB81] suggests this can account for various patterns that have been observed in development.

3. **Chemical**. The basis for morphogenesis, as discussed in chapter two.

4. **Electrical**. Fraser [FP90] showed that electrical activity can affect the formation of synapses during the development of neural structures.

Fleischer [FB93] simulated the genetic, mechanical and chemical influences on a system of discrete cells. Each cell determined its next state from its previous state and the state of its local environment using ordinary differential equations (ODE). Each cell then determined its behaviour from its next state, also using ODEs. Possible behaviours include moving, adhering to neighbouring cells, diffusion of a chemical morphogen, cell division or cell death.

Figure 3.1 shows two cell arrangements that the simulation produced. The simulation starts with two types of cell. One emits the green morphogen and seeks to move to areas of high green concentration, the other emits and seeks the red morphogen. Whether a cell divides, moves or dies is a function of the morphogen concentrations.

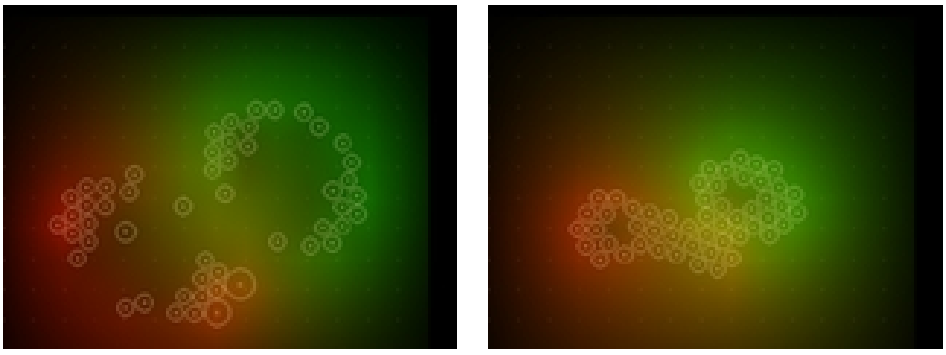


Figure 3.1: Simulated cell development

Images courtesy of Fleischer and Barr

The presence of rings in the pattern surprised Fleischer, who concluded that it was neither trivial to predict the results of any given genome, nor to design a genome to create a particular pattern.

3.2 Designing a genome to create a particular pattern

The unsolved problem of Fleischer's simulation — that of how to design a genome to form a particular pattern — was partially solved by Eggenberger, Liu and Miller using evolutionary algorithms.

3.2.1 Evolutionary algorithms

The power of biological evolution is easy to appreciate; engineers have singularly failed to match the complexity or perfection of the design of the human body. However, in our haste to champion the successes of biological evolution, it is often easy to ignore the huge populations and generations (approximately six billion humanoid forms in this generation cycle, requiring four-thousand million years to generate from simple multi-cellular organisms) that were required for these successes to be possible. With today's resources it is not possible to evolve anything to a degree comparable to the complexity of a humanoid form. To the contrary, the number of input combinations in a truth-table representation of a combinational digital circuit increases exponentially with the number of inputs [Mil00] so that thirty-four million cycles would be required to try every possible output from a comparatively small twenty-five one-bit input circuit.

A large family of algorithms for evolution has been developed over the years, the majority of which are some variant of four separate strains:

1. **Evolutionary programming (EP)** involves the random creation of a population of candidate solutions, each a finite state machine. Mutation is achieved by randomly adding or deleting states, creating or reassigning existing transitions. Stochastic tournament is used to select the fittest solutions. There is no requirement that the population size remain constant, nor that any parent create a fixed number of children [FOW66].

2. **Evolutionary strategies (ES)** were originally designed to find function minima from large multi-variate functions. The candidate solutions evaluated are thus real valued vectors. Evolutionary Strategies use any size population of candidates, mutation and crossover [BS02].

3. **Genetic algorithms (GA)** involves the random creation of a population of "chromosomes": strings of bits that describe a particular solution. A selection of parents is determined by means of a fitness test, and then amalgamated via crossover or mutation to create a number of children. From this new population another set of parents is selected[DM91].

4. **Genetic programming (GP)** uses an application-specific function set arranged in parse trees to represent a particular solution. Unlike GAs, GP does not use mutation. Instead it relies upon a form of crossover between the most successful individuals of a population, selecting random branches of the parse tree of a solution and swapping them with branches from the parse tree of a different solution [BFKN98].

Another characteristic of evolving systems concerns open-endedness. When the fitness criterion is imposed by the user in accordance with the task to be solved, one attains a form of guided evolution. This is to be contrasted with open-ended evolution occurring in nature, which admits no externally-imposed fitness criterion, but rather an implicit, emergent and dynamic one (that could arguably be summed up as survivability) [Shi97].

3.2.2 Eggenberger's evolution-development simulation

Of the four influences on a developing cell, Eggenberger [Egg97] simulated two: genetic and chemical. Each cell could divide, die, or change state depending on its local environment and its present state. An artificial genome programmed into each cell governed this behaviour. A partially-closed GA was used to evolve this genome, with each solution being evaluated against two fitness criteria:

1. The size of the developed system after cell division has stopped, compared to an arbitrary ideal size.
2. The symmetry of the developed system about the x-axis.

Figure 3.2 shows some of the 3D multi-cellular organisms developed on Eggenberger's simulation.

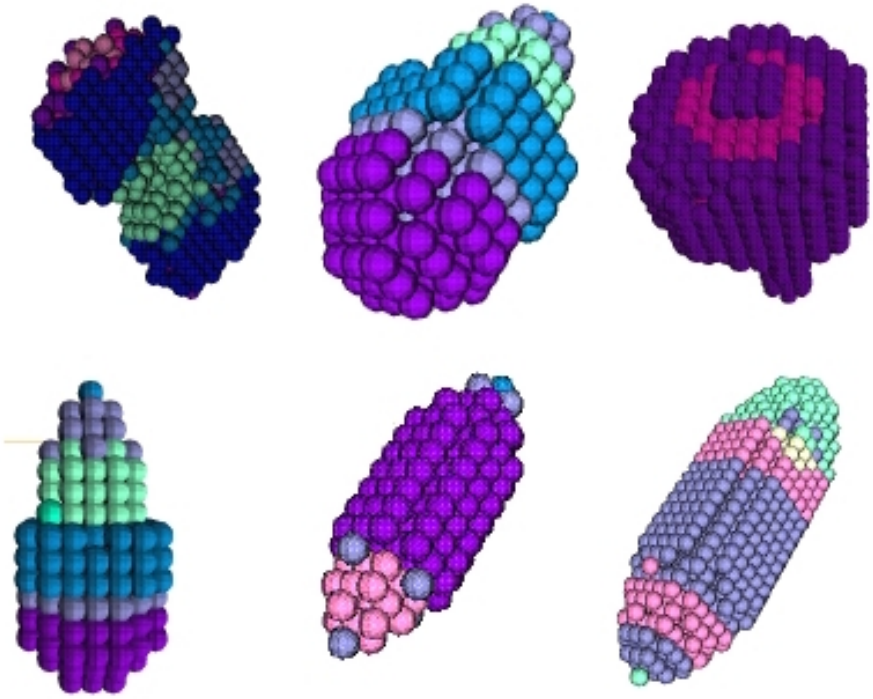


Figure 3.2: Evolved developed forms

Images courtesy of Eggenberger

3.3 Wolpert's French flag

In 1969 Lewis Wolpert [Wol69] proposed that the arrangement of different types of biological cell arises from a combination of intercellular interactions and cellular responses to the chemical gradients formed by the radial diffusion of chemical morphogens from cells. He likened this process to one of growing a simple French flag pattern. Thus the standard for bio-inspired developmental biologists was defined: to be able to develop a French flag pattern of cells from null initial conditions and to be able to repair a French flag pattern from a corrupted pattern.

3.4 Miller's French flag

Of the four influences on a developing cell, Miller [MB03] chose to model just one: the local and global chemical communications between cells. His model used a simulation world that contained 256 cells. The world was responsible for diffusing a single morphogen about one or many cell sources. Each cell used the concentration of the morphogen at its location and the colour of the cells immediately surrounding it to determine both its own colour and whether or not to emit the morphogen itself.

Each cell obeyed the same programming in determining its behaviour, this being coded as a Cartesian genome.

3.4.1 Cartesian genomes

Cartesian Genetic Programming (CGP) is a means of describing a circuit diagram as a series of numbers, termed the genome. CGP is Cartesian in that each node of the described circuit is addressed in a Cartesian co-ordinate system. Each gene of the genome describes a single component, its type and connections to other components.

In [LMT04] the genotype consists of 40 genes, each of which describes the process

and feed-forward interconnections of a two-input one-output boolean process. See figure 3.3 for an example.

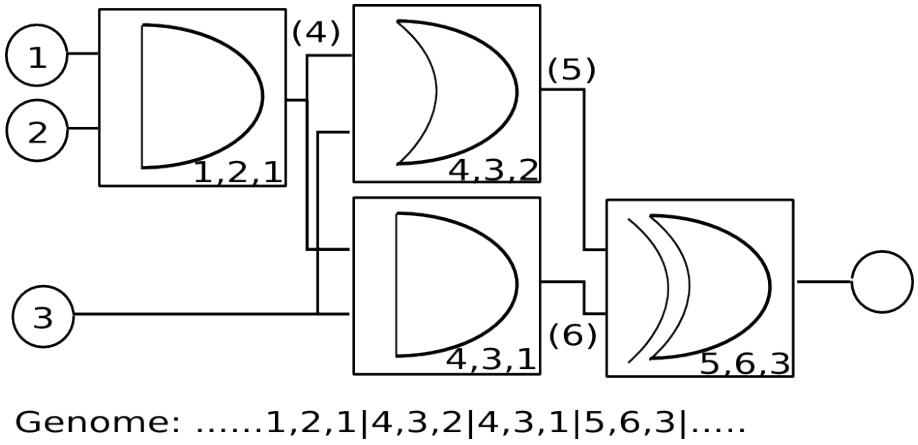


Figure 3.3: Genome to boolean logic mapping

Listing 3.1 is the pseudo-code of a software simulation of the execution of a circuit described by a Cartesian genome.

```
1 For each gene of the genome:
2         Create a row in the table with fields for input and output references and
              values, and component function.
3
4 For each circuit execution:
5         Where the input references match those of the circuits inputs, populate the
              input values.
6         Execute every row that has completed input values.
7         Where the output references of the completed rows match the input references of
               uncompleted rows, populate the input values of those rows.
8         Repeat until all rows have been executed.
```

Listing 3.1: Simulating the execution of a circuit described by a Cartesian genome

3.4.2 The simulation world

Each cell is potentially a chemical point source. The diffusion of this chemical across the array of cells is done at every time-step in accordance with the formula:

$$C_{i,j,t+1} = \frac{1}{2}C_{i,j,t} + \frac{1}{8}\sum_{k,l \in N} C_{k,l,t} \tag{3.1}$$

Where $C_{i,j}$ is the concentration of the chemical at row i and column j in the array.

The neighbourhood N consists of the four cells nearest to (i, j). In the absence of any live neighbours, a cell in Miller's simulation will die at the next time-step. In the presence of live neighbours, a dead cell will become alive at the next time-step. Only live cells may emit morphogens. The concentration of the chemical at each cell determines whether that cell stays alive or dies, and what colour it should output.

3.4.3 The evolution algorithm

Miller used a genetic algorithm called *HereBoy* [Lev00]. HereBoy combines features from genetic algorithms and simulated annealing, and uses a population of two to avoid the necessity to sort fitness values as follows:

1. One parent mutates at a rate calculated from equation (3.2), to form a different child.
2. The fitness of the child is compared to the fitness of the parent.
3. If the fitness is better than the parent, the child replaces the parent as the parent of the next generation.
4. If the fitness is worse than the parent, the child replaces the parent according to a probability calculated from equations (3.3)

$$p(mutate) = \begin{bmatrix} p(mutate)_{max} \\ p_m(1 - \frac{currentfitness}{bestfitness}) \\ p(mutate)_{min} \end{bmatrix} \tag{3.2}$$

$$p(reverse) = \begin{bmatrix} p(reverse)_{max} \\ p_r(1 - \frac{currentfitness}{bestfitness}) \\ p(reverse)_{min} \end{bmatrix} \tag{3.3}$$

The pseudo-code of the application of HereBoy to the design of Miller's morphogenetic French flag can be seen in listing 3.2.

```
1 Create a random 40-gene genome where each gene is a 3-integer code for the function and
      input set of one component.
2
3 Do till fitness = 100%:
4         Develop the world over 9 time-steps using this genome in each cell
5         For times 6,7,8,9 evaluate the fitness of the solution by comparing the pattern
              developed to that of the French flag.
6         If this genome has a higher fitness than that of its predecessor, keep it
              except in circumstances determined by the probability p_reverse.
7         Calculate p_mutate and p_reverse
8         Mutate a number of genes of the genome, selected according to the probability
              p_mutate
9         Repeat.
```

Listing 3.2: Simulating the execution of a circuit described by a Cartesian genome

3.4.4 Results

Figure 3.4 shows a French flag pattern being developed from a single cell. Figure 3.5 shows its corresponding morphogen concentrations. Figure 3.6 shows a French flag pattern repairing itself from corrupt initial conditions.

This algorithm was capable of designing small self-assembling French flag patterns and repairing minor damage to them.

3.5 Liu's French flag

In contrast to Miller's work, developing biological systems use more than one chemical morphogen. As demonstrated in chapter one, the interaction of multiple morphogens can create irregular localised concentrations. Miller's work, limited to one morphogen, can only form overlapping radial distributions about each point source. Liu [LMT04] proposed a solution to this limitation that did not require the significant complexity of simulating multiple interacting chemicals.

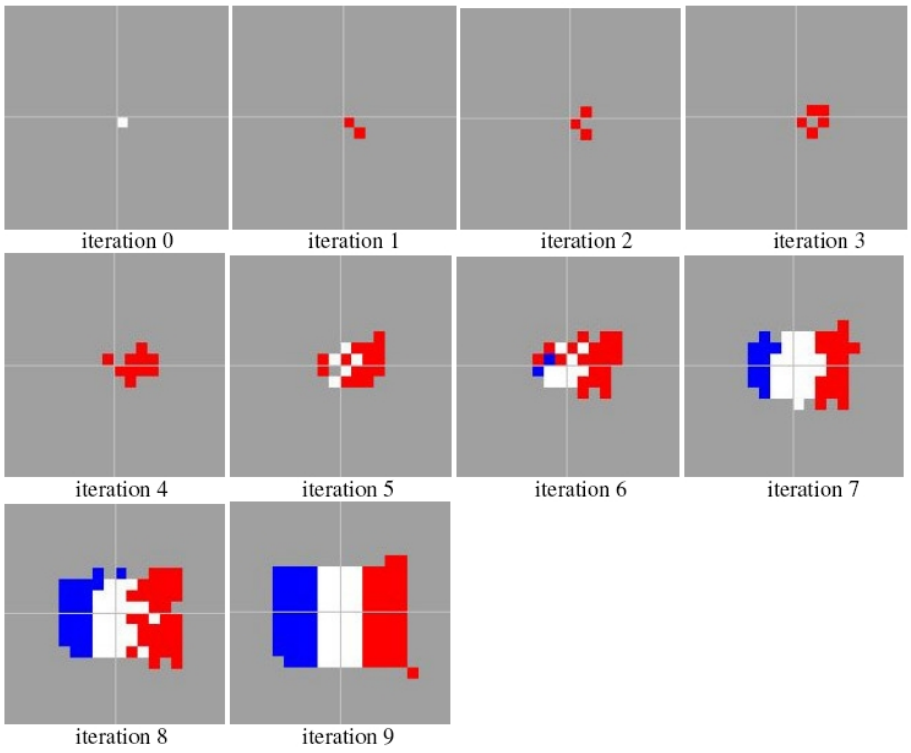


Figure 3.4: A French flag pattern developed from a single cell

Images courtesy of Miller; nb in the author's images "iteration n" is labelled "t = n"

3.5.1 A new diffusion model

At a particle level, diffusion is the movement of each particle from its present location in a random direction. Over time, collisions amongst particles in the denser areas will ensure that any local permutations in concentration disappear. Given a constantly transmitting point source and a homogeneous medium, diffusion will create a radial distribution with its origin at the source.

By increasing the probability a particle will move in a particular direction over another, the chemical of which the particle is a part will no longer diffuse in a radial distribution. Instead, a heterogeneous medium is formed with each discrete location effectively a different concentration. Evolving this medium alongside the cellular computation algorithms creates a solution of which localised spatial patterns of chemical concentrations is a part.

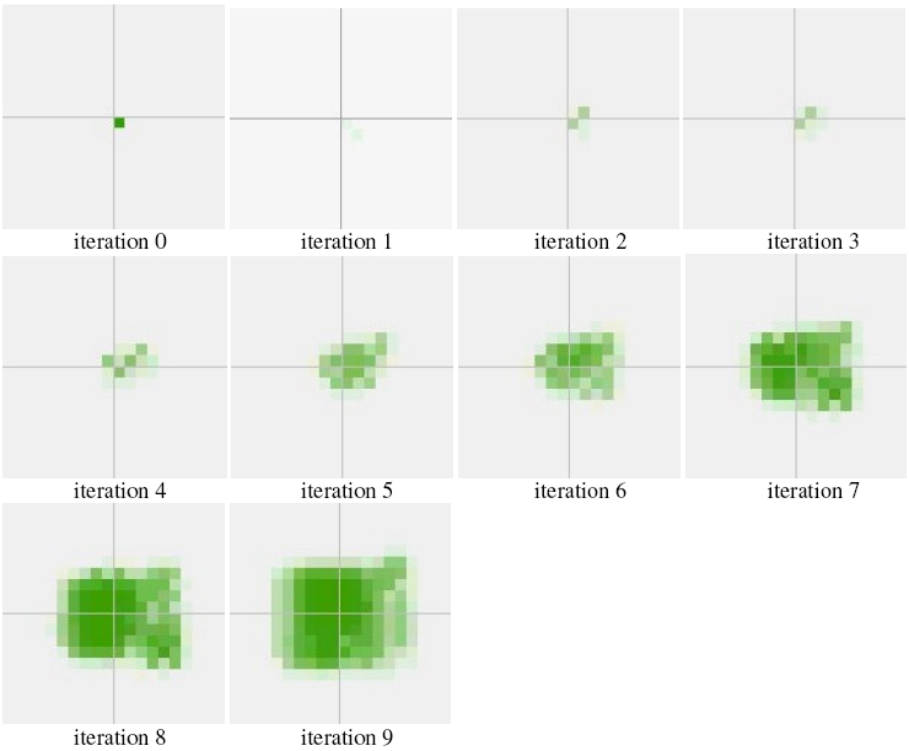


Figure 3.5: The corresponding morphogen concentrations

Images courtesy of Miller

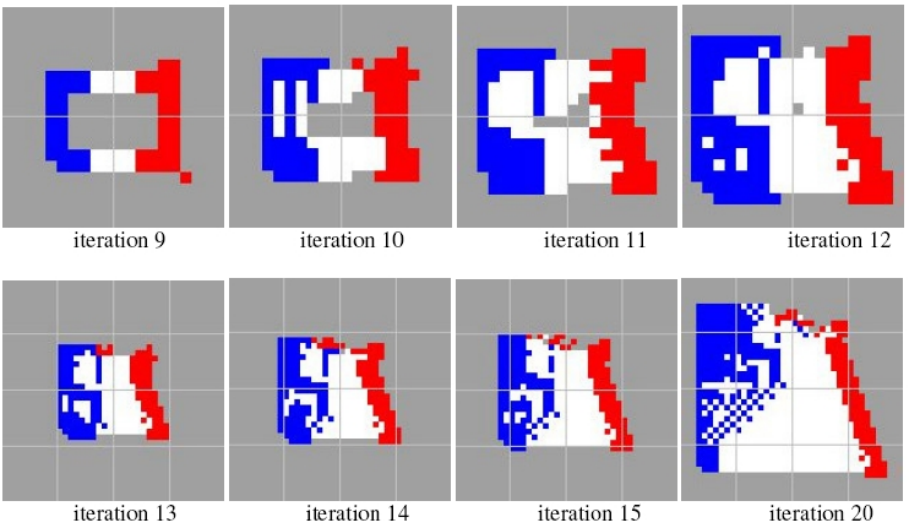


Figure 3.6: A French flag pattern developed from a corrupt flag

Images courtesy of Miller

The morphogen concentration at each point in the array is described as the sum of two components: a concentration common to all cells on the same row which is diffusing along the y-axis and a similar component diffusing along the x-axis. The rate of diffusion along each axis is governed by a series of eight-bit values that determine what proportion of the concentration at each point along each axis moves left or right, up or down. These eight-bit values are determined by the evolutionary algorithm. The development-evolution pseudo-code differs little from that already detailed. Liu used 48 genes instead of Miller's 40, and simulated the flag assembly on a smaller, six by six, world of cells.

3.5.2 Results

Figure 3.7 shows a French flag pattern being developed from null initial conditions and figure 3.8 shows the simulation attempting to repair itself from a corrupt starting pattern.

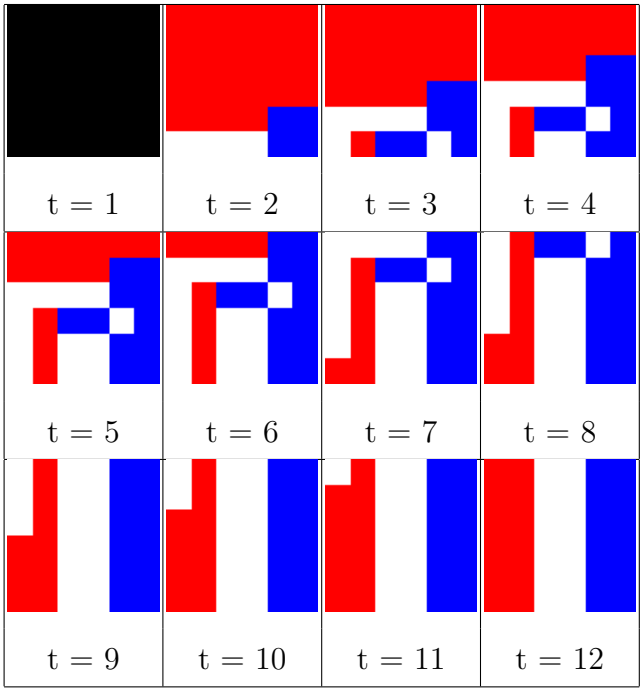


Figure 3.7: A French flag pattern developed from null initial conditions

A simple program designed to test all possible starting patterns showed that this model is capable of repairing the French flag pattern in the event that up to 25% of

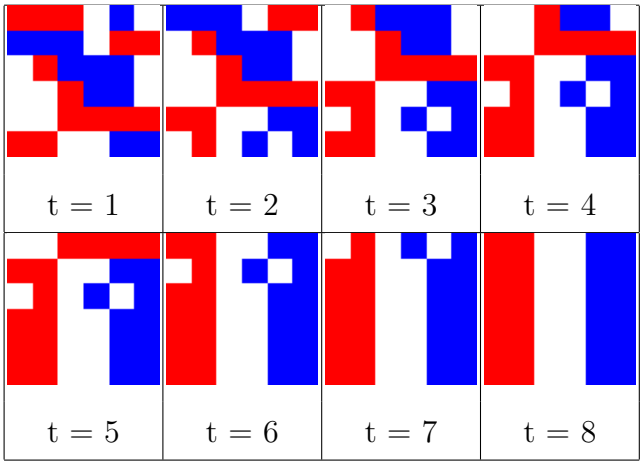


Figure 3.8: A French flag pattern repaired from corrupt initial conditions

the pattern is corrupt.

3.6 Conclusions

Fleischer proved the potential of simulated cellular development with the development of complicated cell patterns. His work also showed that it is difficult to design a developing system to create particular patterns — a problem which has affected all such studies since. Eggenberger demonstrated the application of a partially-closed genetic algorithm to the design of 3D cellular arrangements limited only in size of arrangement and its symmetry about the x-axis. Miller and Liu used a closed genetic algorithm to design a self-assembling, self-repairing simple French flag pattern. Liu's algorithm was capable of repairing up to 25% corruption to the flag pattern. In the next chapter we will introduce a model of morphogenesis based on cellular automata, then determine the conditions required for it to be able to repair 100% corruption to the pattern.

Chapter 4

Mimicking morphogenesis with convergent cellular automata

> When the sum of the parts adds up to more than those parts then that extra 'being' — the something from nothing — is arising from a field of many interacting smaller pieces [Kel95].

The complete human body consists of approximately 100 trillian cells. Morphogenesis is responsible for co-ordinating their differentiation from their original stem cell form. It achieves this by means of long-range diffusions of chemical morphogens. The community effect uses the diffusion of a different family of chemical signals with a much shorter range. See chapters two and eight for more details of these techniques.

Whilst the imitation of this technique to achieve similar resilience is the aim this chapter addresses, to imitate regeneration on a similar scale is impractical. Thus one consideration is which communication technique — the community effect or morphogenesis — is more appropriate on a cellular automata of 10 to 100 cells. Morphogenesis is principally responsible for cell differentiation; the community effect for localised cohesion. Differentiation is a necessary part of systems design, but on systems of 10 to 100 cells the community effect has the more appropriate scale. A

typical cellular automata (CA) consists of homogeneous, locally-interacting cells, but the cumulative effect of repeated local interactions can mimic long range interactions. A CA that converges to a single fixed point, regardless of its initial conditions is demonstrating the same resilience shown by regeneration of the liver, but at a different scale and on a different platform.

The following chapter will introduce cellular automata then determine the necessary CA design such that it will always return to form the same pattern using an equivalent matrix model of CA and a functional analysis of a matrix metric space.

4.1 Cellular automata, their classification and design

Cellular automata (CA) systems are dynamic systems in which space and time are discrete [Hoh68]. CAs consist of a number of identical cells arranged in an n-dimensional array. Each cell can be in one of a number of states. The next state of each cell is determined at discrete time intervals according to the current state of the cell, the current state of the neighbouring cells and a next state rule that, in the case of homogeneous CA systems, is identical for each cell.

Some famous CA include the Turing machine [Tur50b], Neumann's self-replicating universal constructor [Neu66], Conway's game of life [Gar70] and Langton's ant [Lan86, Gal93, Gal98].

It is difficult to predict the state of a typical CA after a significant number of iterations. It is even more difficult to determine the next state rule directly from a desired state after a significant number of iterations. Some efforts have been made to classify CA behaviour and to study the correlation between this behaviour and the form of its next state rule.

The behaviour of different CA can be separated into two criteria:

1. The form of the output. Is the CA capable of forming simple, chaotic or complex structures?

2. The dependence of the output upon the input. To what degree does changing the input vary the output, where the input is the initial state of each cell?

Wolfram used a one-dimensional variant of Neumann's cellular automata with each generation of cells being displayed below its parent. Each cell has a two-state alphabet, that is it can be in one of two states that are displayed as black and white. The next state of each cell is determined by one of 256 possible rules whose inputs are the states of its two neighbours and its own state [Wol82]. Wolfram grouped each of the 256 possible rules into four classes, described below, according to their resultant form.

4.1.1 Class one cellular automata

Class one cellular automata develop after a finite number of time steps from almost all initial states to a state in which all cells have the same value. After sufficient iterations, any information in the initial state has been destroyed. Figure 4.1 shows the development of a class one CA from a random initial state to a homogeneous zero state, the CA array on the x-axis and the successive time steps displayed down the y-axis.

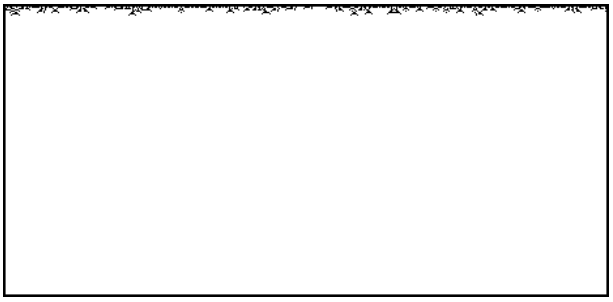


Figure 4.1: Rule 36. A member of the class one set. This rule will always reach a homogeneous state

Of the 256 possible rules of a one-dimension two-state CA, 216 are members of the class one set.

4.1.2 Class two cellular automata

Class two cellular automata generate simple structures from particular initial cell state sequences. Changes to the initial state almost always affect the final state. Figure 4.2 shows the development of a class two CA from an initial random state.

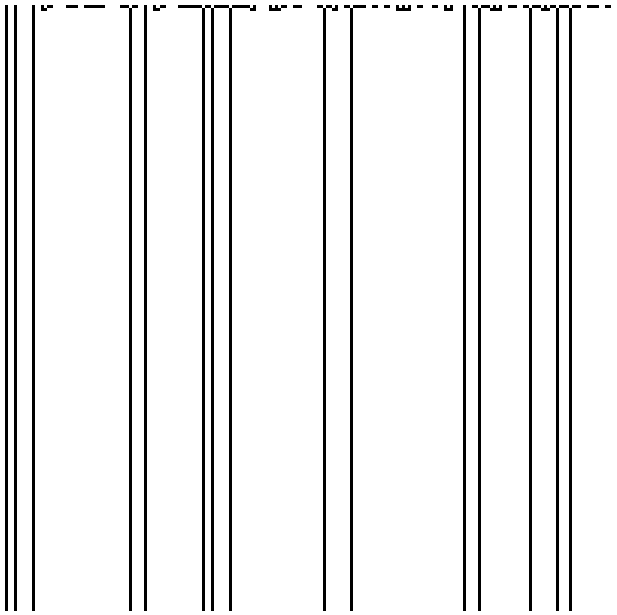


Figure 4.2: Rule 36, a member of the class two set. This rule displays sensitivity to initial conditions.

Of the 256 possible rules of a one-dimension two-state CA, 24 are members of the class two set.

4.1.3 Class three cellular automata

Class three cellular automata generate aperiodic chaotic patterns from almost all possible initial states. Figure 4.3 shows the development of a class 3 CA from an initial state of one live cell.

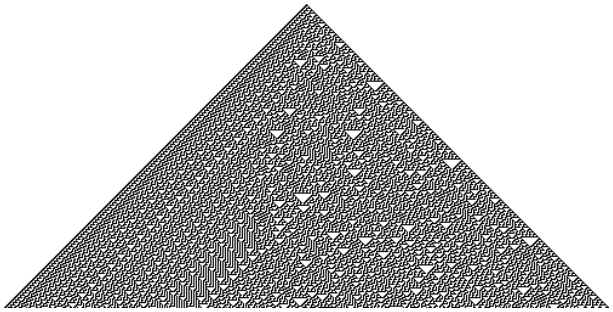


Figure 4.3: Rule 30, a member of the class 3 set. This rule displays aperiodic chaotic patterns and is used as part of the random number generator in the software package Mathematica.

Of the 256 possible rules of a one-dimension two-state CA, ten are members of the class 3 set.

4.1.4 Class four cellular automata

It has been hypothesised that class four CA are the only CA capable of being computationally universal. Over time, given the right initial conditions, a class 4 CA will behave in part ordered and in part chaotic. The study of this behavioural model is called complexity theory. Figure 4.4 shows the development of a class four CA from a random initial state.

Of the 256 possible rules of a one-dimension two-state CA, six are members of the class four set.

4.1.5 Reversible cellular automata

A CA is reversible if, given the rules it obeys, for every possible CA state we can deduce what the previous state of the CA was. Thus there is a one-to-one mapping from each possible state to its predecessor. If a CA is reversible it is possible to design the rules it must obey to be in particular states at particular times by induction.

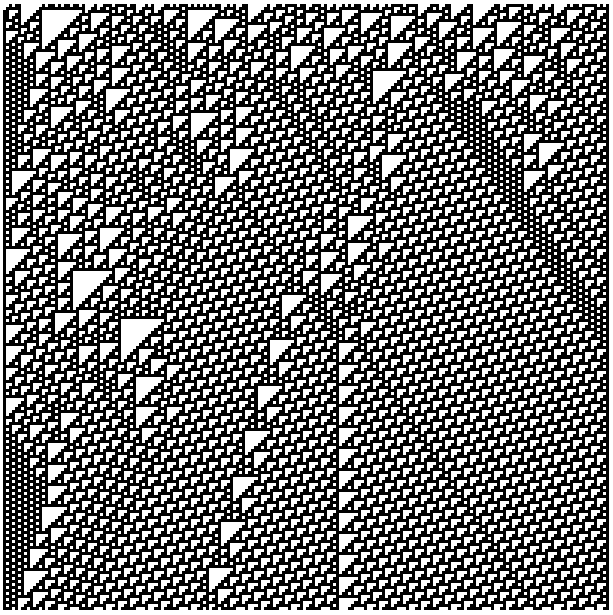


Figure 4.4: Rule 46, a member of the class four set. This rule displays complex behaviour.

A CA is second-order if each cell determines its next state based not only on the states of its neighbours at time $t-1$ but also on their state at time $t-2$. If a CA is second-order and there is a one-to-one mapping between the state of a cell at $t-2$ and its next state then it is reversible [Wol02].

Another way to make a CA reversible is to partition it into Margolus neighbourhoods [TM90]. The CA is split into groups of neighbourhoods of two by two cells. As time progresses, neighborhoods are alternate between even and odd coordinates so if at $t = 0$ a neighbourhood consists of cells (1,1),(1,2),(2,1) and (2,2), at $t = 1$ the neighbourhood would become (2,2),(2,3),(3,2),(3,3). Each cell in the neighbourhood determines its next state based on the state of the other cells in its neighbourhood.

4.1.6 Langton's lambda parameter

Langton [Lan90] developed the Lambda parameter, λ, as a means of categorising different cellular automata. It is a measure of the distribution of state transitions within the next state rule. This can be used to predict the complexity generated by

a CA.

$$\lambda(x) = \frac{\text{No. of rules that lead to x}}{\text{No. of rules}} \tag{4.1}$$

For chaotic systems the λ region is centered about 0.5. For complex systems the λ region is centered about 0.25.

Another measure used to categorise CA is the mutual information between cells, C, where the mutual information is defined as the sum of the entropy of the individual cells minus the entropy of the cells as a collective [Lan90].

$$I = \sum_{c=1}^{C} \sum_{r=1}^{R} p(x_{c,r}) \log_2 p \left(\frac{1}{x_{c,r}} \right) - \sum_{j=1}^{CN} p(X_j) \log_2 \left(\frac{1}{p(X_j)} \right) \tag{4.2}$$

Where:

1. C is the number of cells
2. R is the size of the alphabet of each cell
3. $p(x_{c,r})$ is the probability of cell x_c being in state r
4. CN is the number of possible patterns the automata can form
5. $p(X_j)$ is the probability of the automata X being in state j

This equation is formed by calculating the entropy of the array of cells, then subtracting the sum of the entropies of each individual cell.

4.1.7 Models of independence

Two criteria determine the sensitivity to initial conditions of a CA, as defined by the difference in entropy as the CA develops.

1. The dependence of each rule's output on its inputs. Class 1 CA saturate to an homogeneous entropy minima after sufficient time steps. This is due to the independence of most, if not all, of the outputs of each rule on their respective input combinations. For instance rule zero, whose transitions will result in a zero state, irrespective of whether the inputs are one or zero.

2. The dependence of each rule's output on the position of each information source relative to the active cell. The edges of a bounded CA are entropy minima. Thus if information only propagates in one direction the information of the CA will be forced to an edge then removed from the CA. Figure 4.5 shows the development of a CA that transconducts information to the right edge until a homogeneous stable state is reached.

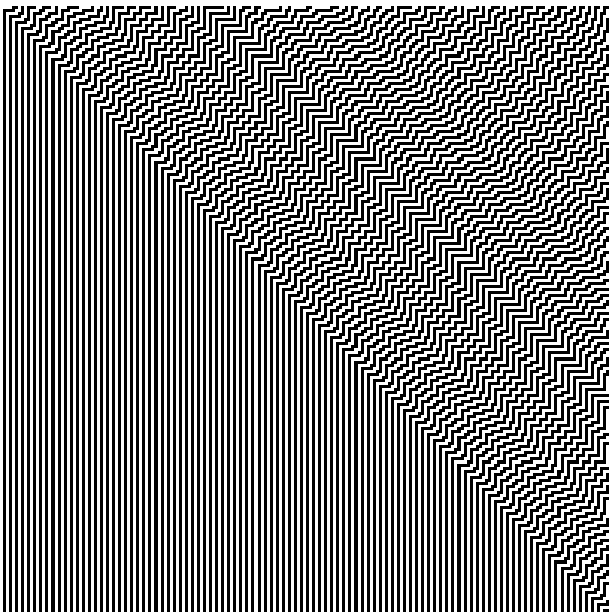


Figure 4.5: Rule 15, the output of this rule is independent of its inputs, because the information moves from the centre to the right edge

4.1.8 Mean field theory

The probability of any cell being in the state one, p_1 is the sum of the probabilities of any group of adjacent cells forming an input combination that maps to a next

state of one. If $n_0(i)$ is the number of cells needed to be in state 0, $n_1(i)$ the number of cells needed to be in state one for a next state rule i then p_1 is given by [GL95]:

$$p_{1,t+1} = \sum_{i \in \text{Rules}_{\to 1}} p_{1,t}^{n_1(i)} (1 - p_{1,t})^{n_0(i)} \tag{4.3}$$

This assumes there is no correlation between the cell states at time, t. This assumption can be relaxed using Markov state modelling or local structure theory [GVK87], making the model more precise.

The statistical estimates created using the lambda parameter or mean field theory can be used to approximate CA next state rules that result in a desired CA complexity. They are commonly used to refine the search field of genetic algorithms.

The stated aim of this research is to be able to design CA to converge to a specific pattern with a greater certainty than offered by available statistical tools and without the overhead of genetic algorithms. Reversible CA, though trivial to design, are restricted to converging towards a small set of possible patterns.

The following section will use matrix algebra to determine the requisite conditions for a CA to always converge to the same state, regardless of its initial conditions.

4.2 An equivalent matrix model

For purposes of clarity, we will start with a one-dimensional CA of identical cells that use information from their nearest neighbours (to the east and west of itself) to compute their next state. Each cell computes its next state at the same time and does so at each discrete time step. Let us index each cell (c_i) $i \in \{1..N\}$, then describe the state of each cell at time t with an integer, $c_{i,t}$ and the pattern of the entire array as a vector, $\overline{C}_t$.

If $\overline{C}_0$ is the initial pattern of cell states, $f(\overline{C}_0)$ is its subsequent pattern after one

time step, and $f(f(\overline{C}_0))$ or $f^2(\overline{C}_0)$ is its pattern at $t = 2$; where f describes the transition from one iteration to the next.

An additive CA is such that the next state of any cell is the result of assigning a coefficient (v, w, x) to each member of the neighbourhood set, adding the states of those in its neighbourhood to a constant k, and then applying a modulus function. The modulus function $\mod r$ defines the size of the alphabet of the CA. Let us now define a simple function that describes the transition for an additive CA from one state to the next:

$$c_{i,t+1} = vc_{i-1,t} + wc_{i,t} + xc_{i+1,t} + k \mod r \tag{4.4}$$

A transition function for the entire array can be formed from (4.4) such that

$$f(\overline{C}_t) = \mathbf{A}\overline{C_t} + \overline{K} \mod r \tag{4.5}$$

Where $\overline{K}$ is a constant and the transition matrix, $\mathbf{A}$, takes the form (when $N = 6$):

$$\mathbf{A} = \begin{bmatrix} w & x & 0 & 0 & 0 & 0 \\ v & w & x & 0 & 0 & 0 \\ 0 & v & w & x & 0 & 0 \\ 0 & 0 & v & w & x & 0 \\ 0 & 0 & 0 & v & w & x \\ 0 & 0 & 0 & 0 & v & w \end{bmatrix} \tag{4.6}$$

4.3 One dimensions, from first to last state

4.3.1 The transition function from first to last state

Iterating (4.4) twice gives (4.7), three times gives (4.9) and n times gives (4.11).

$$\overline{C}_{t+2} = \mathbf{A}\left(\mathbf{A}\,\overline{C_t} + \overline{K}\right) + \overline{K} \mod r \tag{4.7}$$

$$= \mathbf{A}^2\,\overline{C_t} + \mathbf{A}\,\overline{K} + \overline{K} \mod r \tag{4.8}$$

$$\overline{C}_{t+3} = \mathbf{A}\left(\mathbf{A}\left(\mathbf{A}\,\overline{C_t} + \overline{K}\right) + \overline{K}\right) + \overline{K} \mod r \tag{4.9}$$

$$= \mathbf{A}^3\,\overline{C_t} + \mathbf{A}^2\,\overline{K} + \mathbf{A}\,\overline{K} + \overline{K} \mod r \tag{4.10}$$

$$\overline{C}_{t+n} = \mathbf{A}^n\,\overline{C}_t + \mathbf{A}^{n-1}\,\overline{K} + \mathbf{A}^{n-2}\,\overline{K} + \mathbf{A}^{n-3}\,\overline{K} \cdots + \mathbf{A}^1\,\overline{K} + \overline{K} \mod r \tag{4.11}$$

Using the sum of the geometric series equation:

$$f(x) = \sum_{j=0}^{k} r^j = \frac{1 - r^{k+1}}{1 - r} \tag{4.12}$$

we can simplify (4.11) to:

$$\overline{C}_{t+n} = \mathbf{A}^n\overline{C_t} + \sum_{p=0}^{n-1} \left(\mathbf{A}^p\right)\,\overline{K} \mod r \tag{4.13}$$

$$\overline{C}_{t+n} = \mathbf{A}^n\overline{C_t} + \frac{\mathbf{I} - \mathbf{A}^n}{\mathbf{I} - \mathbf{A}}\,\overline{K} \mod r \tag{4.14}$$

4.3.2 The conditions for convergence

Thus for the output, $\overline{C}_{t+n}$ to be independent of initial state, $\overline{C}_{t=0}$, the term $\mathbf{A}^n\,\overline{C}_{t=0}$ must equal zero. Given that n is greater than the dimensions of the CA, $\mathbf{A}^n\,\overline{C}_{t=0}$ will equal zero if $\mathbf{A}$ is either an upper-triangular or lower-triangular matrix. Thus either v or x and w of equation (4.4) must equal zero for the CA to converge.

To prove this is the only form **A** can take for any CA from first principles we need to use functional analysis and matrix metric spaces [1]

4.4 Metric spaces

The study of metric spaces allows mathematicians to study the continuity and convergence of functions. Before we can use them to study convergence on a CA metric space, we need to introduce a few terms.

Definition 1 A **space** is a non-empty set, $\mathbf{X}$. The points of a space are the elements of the set, $x \in \mathbf{X}$. An example space is the two-dimensional euclidean space $\mathbb{R}^2$. The space $M_{n,m}(\mathbb{R})$ is the set of all possible $n \times m$ matrices. □

Definition 2 A **metric**, d is the distance between two points $x, y \in \mathbf{X}$ of a space.

A metric obeys the following four axioms:

1. $d(x, y) = d(y, x) \ \forall \ x, y \in \mathbf{X}$. The distance from x to y is the same as the distance from y to x.
2. $d(x, y) > 0 \ \forall \ x, y \in \mathbf{X}$ and x $\neq y$. A distance must be a real-valued non-negative distance.
3. $d(x, y) = 0 \Leftrightarrow x = y \ \forall \ x, y \in \mathbf{X}$.
4. $d(x, z) \leq d(x, y) + d(y, z) \ \forall \ x, y, z \in \mathbf{X}$. Known as the triangular inequality, the distance between any two points is equal or greater if travelled via another point.

A commonly used metric is the euclidean distance function:

[1] This proof was derived, in part, from the lecture notes of Dr Dirk Schutz [Sch05].

$$\|\overline{x} - \overline{y}\| = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{4.15}$$

Because the square of $x_i - y_i$ is symmetrical about $x_i = y_i$, only equal to zero at this point, real-valued and non-negative the euclidean distance meets the first three axioms. To show that it meets the triangular inequality axiom, let u be the vector $\overline{y} - \overline{x}$ and v be the vector $\overline{z} - \overline{y}$.

$$\|\overline{u} + \overline{v}\|^2 = (\overline{u} + \overline{v}) \cdot (\overline{u} + \overline{v}) \tag{4.16}$$

Using the euclidean inner product rule we get:

$$\begin{aligned} \|\overline{u} + \overline{v}\|^2 &= \overline{u} \cdot (\overline{u} + \overline{v}) + \overline{v} \cdot (\overline{u} + \overline{v}) && (4.17)\\ &= \overline{u} \cdot \overline{u} + 2(\overline{u} \cdot \overline{v}) + \overline{v} \cdot \overline{v} && (4.18)\\ &\leq \|\overline{u}\|^2 + 2|\overline{u} \cdot \overline{v}| + \|\overline{v}\|^2 && (4.19) \end{aligned}$$

Now using the Cauchy-Schwartz inequality ($|\overline{u} \cdot \overline{v}| \leq \|\overline{u}\|\|\overline{v}\|$).

$$\begin{aligned} \|\overline{u} + \overline{v}\|^2 &\leq \|\overline{u}\|^2 + 2\|\overline{u}\|\|\overline{v}\| + \|\overline{v}\|^2 && (4.20)\\ &\leq (\|\overline{u}\| + \|\overline{v}\|)^2 && (4.21)\\ \|\overline{u} + \overline{v}\| &\leq \|\overline{u}\| + \|\overline{v}\| && (4.22) \end{aligned}$$

Thus the euclidean distance function meets the triangular inequality axiom. □

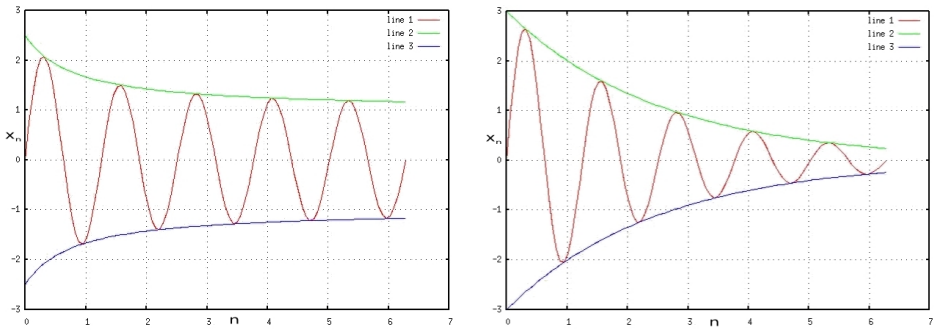
Definition 3 To be a **metric space**, $(\mathbf{X}, d)$, d must be a metric on the space $\mathbf{X}$ □

Definition 4 A sequence of points x_n converges in $(\mathbf{X}, d)$ to a point $x \in \mathbf{X}$ if for any real number $\varepsilon > 0$ there is an integer $N > 0$ so that:

$$d(x_n, x) < \varepsilon \; \forall \; n > N \tag{4.23}$$

□

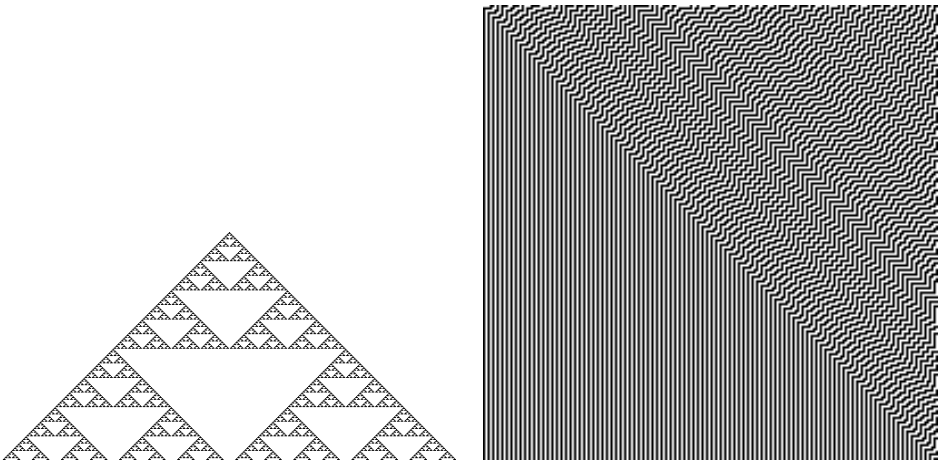
Figure 4.6 shows a non-converging sequence and a converging sequence in the metric space $(\mathbb{R}, d)$.



$y(x) = \frac{x+2}{x+0.8}\sin(5x)$ $\qquad$ $y(x) = 3\exp(-0.4x)\sin(5x)$

Figure 4.6: Convergence in $(\mathbb{R}, d)$

Figure 4.7 shows converging and non-converging sequences in the vector metric space $(\mathbf{X_n}|x_i \in \{0, 1\})$, where a black pixel represents a 1, a white pixel represents a 0 and the y-axis shows the progression of time (from top to bottom).



Wolfram's rule 22 CA $\qquad$ Wolfram's rule 15 CA

Figure 4.7: Convergence in $(\mathbf{X_n}|x_i \in \{0, 1\})$

Definition 5 A transform $f : \mathbf{X} \to \mathbf{X}$ on $(\mathbf{X}, d)$ is a contraction mapping if there exists $0 \leq \lambda < 1$ such that:

$$d(f(x), f(y)) \leq \lambda.d(x, y) \; \forall \; x, y \in \mathbf{X} \tag{4.24}$$

where λ is the contraction factor. If $0 \leq \lambda \leq 1$ this is a non-expansive mapping.

A common example of a contraction mapping is the Sierpinski triangle. Listing 4.1 shows the pseudo-code for creating the Sierpinski triangle using the affine transformation $w : \mathbb{R}^2 \rightarrow \mathbb{R}^2$:

$$w\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \tag{4.25}$$

□

Using the values $a = 0.5$, $b = 0$, $c = 0$, $d = 0.5$, $e = 15$ and $f = -8$ this transformation has a contraction factor of 0.5.

Figure 4.8 shows the Sierpinski triangle after 10 iterations.

```
1 Start with an equilateral triangle with a base parallel to the horizontal axis
2 Do forever:
3         Shrink the triangle to half the original height and width, make three copies,
              and position the three shrunken triangles so that each triangle touches the
               two other triangles at a corner
```

Listing 4.1: Sierpinsiki pseudo-code

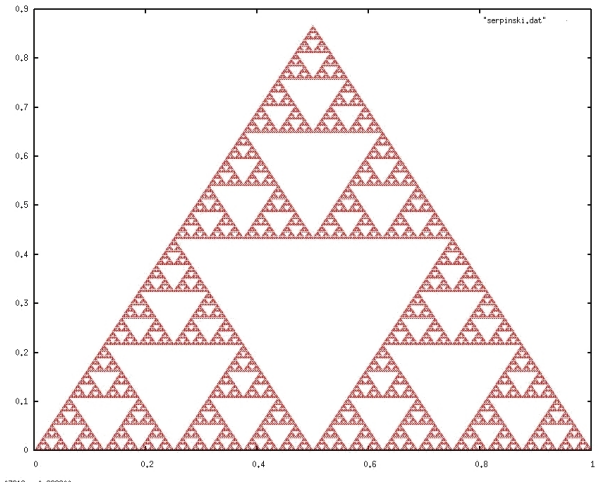


Figure 4.8: The Sierpinski triangle after 10 iterations

Lemma 1 *There can only be one fixed point for each contraction mapping.*

Let us consider the alternative, that two fixed points a and b exist such that:

$$f(a) = a \tag{4.26}$$

$$f(b) = b \tag{4.27}$$

For this to be so

$$d(a,b) = d(f(a), f(b)) \tag{4.28}$$

$$\leq \lambda d(a,b) \tag{4.29}$$

Therefore $d(a,b) = 0$ so the fixed point a equals the fixed point b.

4.5 The Lagrange multiplier

The method of Lagrange multipliers is a means of finding the maxima and minima of multi-variable systems subject to one or more constraints.

Definition 1 The Lagrangian $\wedge(x,y,\zeta)$ of a function $f(x,y)$ constrained by the function $g(x,y) = 0$ is:

$$\wedge(x,y,\zeta) = f(x,y) + \zeta g(x,y) \tag{4.30}$$

□

The stationary points of $\bigtriangledown(x,y,\zeta)$ occur when the gradients of f, $\bigtriangledown_{x,y} f$ and g, $\bigtriangledown_{x,y} g$ are parallel vectors.

This occurs when:

$$\frac{\partial}{\partial x} \wedge (x,y,\zeta) = 0 \tag{4.31}$$

$$\frac{\partial}{\partial y} \wedge (x,y,\zeta) = 0 \tag{4.32}$$

$$\frac{\partial}{\partial \zeta} \wedge (x,y,\zeta) = 0 \tag{4.33}$$

Thus to find the maxima and minima of f constrained by g, we solve the partial differential equations of $\wedge(x, y, \zeta)$ equal to zero.

For example, let us find the maximum of $\{x + y | x^4 + y^4 - 1 = 0\}$, shown in figure 4.9.

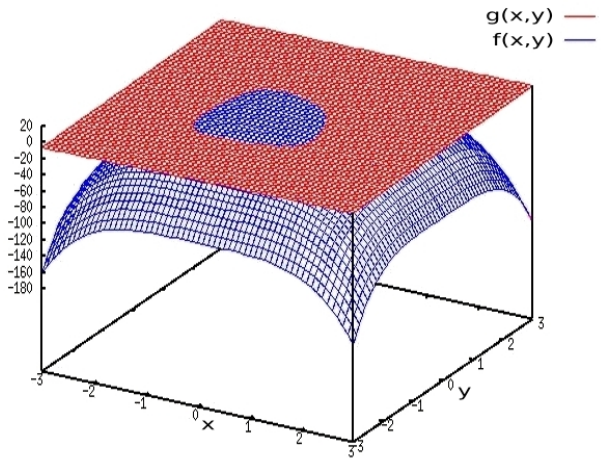


Figure 4.9: Finding the Lagrangian of f constrained by g

The constraint $g : \mathbb{R}^2 \rightarrow \mathbb{R}$ is $g(x, y) = 1 - (x^4 + y^4)$.

Therefore the Lagrangian $\wedge(x, y, \zeta) = x + y - \zeta(x^4 + y^4 - 1)$

Partially differentiating with respect to (x, y, ζ) gives:

$$\frac{\partial \wedge (x, y, \zeta)}{\partial x} = 1 - 4x^3\zeta = 0 \tag{4.34}$$

$$\frac{\partial \wedge (x, y, \zeta)}{\partial y} = 1 - 4y^3\zeta = 0 \tag{4.35}$$

$$\frac{\partial \wedge (x, y, \zeta)}{\partial \zeta} = 1 - x^4 - y^4 = 0 \tag{4.36}$$

Substituting (4.34) and (4.35) into (4.36) gives:

$$\left(\frac{1}{4\zeta}\right)^{\frac{4}{3}} + \left(\frac{1}{4\zeta}\right)^{\frac{4}{3}} = 1 \tag{4.37}$$

$$\zeta = \pm\left(\frac{8}{4}\right)^{\frac{1}{4}} \tag{4.38}$$

So there are two critical points at $\pm(\frac{1}{\sqrt[4]{2}}, \frac{1}{\sqrt[4]{2}})$ and the maximum value is $\frac{2}{\sqrt[4]{2}}$

4.6 The conditions for a cellular automata to converge

Let us define the metric space $(\mathbf{M}(\mathbb{Z}), d)$ where:

1. Every possible pattern on a one-dimensional CA is a member of $\mathbf{M}(\mathbb{Z})$
2. The metric, d is the euclidean distance function

$$\|\overline{C_{t=a}} - \overline{C_{t=b}}\| = \sqrt{(c_{1,a} - c_{1,b})^2 + (c_{2,a} - c_{2,b})^2 + \cdots + (c_{n,a} - c_{n,b})^2} \tag{4.39}$$

   where n is the size of the CA.

Suppose our CA transition function f is:

$$f(C_0) = \mathbf{A}.C_0 + D \mod r \tag{4.40}$$

The euclidean distance $\|C_0 - C_1\|^2$ is:

$$\|C_0 - C_1\|^2 = [c_{0,0} - c_{0,1}\ c_{1,0} - c_{1,1}\ \cdots c_{n,0} - c_{n,1}] \begin{bmatrix} c_{0,0} - c_{0,1} \\ c_{1,0} - c_{1,1} \\ \vdots \\ c_{n,0} - c_{n,1} \end{bmatrix} \tag{4.41}$$

$$= \begin{bmatrix} c_{0,0} - c_{0,1} \\ c_{1,0} - c_{1,1} \\ \vdots \\ c_{n,0} - c_{n,1} \end{bmatrix}^T \begin{bmatrix} c_{0,0} - c_{0,1} \\ c_{1,0} - c_{1,1} \\ \vdots \\ c_{n,0} - c_{n,1} \end{bmatrix} \tag{4.42}$$

Therefore the euclidean distance $\|f(C_0) - f(C_1)\|^2$ is:

$$\|f(C_0) - f(C_1)\|^2 = ((\mathbf{A}C_0 + D) - (\mathbf{A}C_1 + D))^T((\mathbf{A}C_0 + D) - (\mathbf{A}C_1 + D)) \tag{4.43}$$

$$= (\mathbf{A}C_0 - \mathbf{A}C_1)^T(\mathbf{A}C_0 - \mathbf{A}C_1) \tag{4.44}$$

$$= (C_0 - C_1)^T\mathbf{A}^T\mathbf{A}(C_0 - C_1) \tag{4.45}$$

From (4.24), for f to be a contraction mapping, the distance between any two points x and y must be greater than or equal to the distance between $f(x)$ and $f(y)$. To prove that f is a contraction mapping, we need to show that the contraction factor λ is less than or equal to one for any two points in the matrix space. Let us find the maximum value of λ, λ_{max}:

$$\lambda_{max} = \frac{(C_0 - C_1)^T\mathbf{A}^T\mathbf{A}(C_0 - C_1)}{(C_0 - C_1)^T(C_0 - C_1)} \tag{4.46}$$

Let us now solve equation (4.46) for an automata of size $n = 2$.

Let $C_0 - C_1 = \begin{bmatrix} u \\ v \end{bmatrix}$ and

$$u^2 + v^2 = 1 \tag{4.47}$$

This constraint is required so the denominator of (4.46) disappears. Therefore:

$$\lambda_{max} = [u\ v]\mathbf{A}^T\mathbf{A}\begin{bmatrix} u \\ v \end{bmatrix} \tag{4.48}$$

If $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, multiplying (4.48) out gives:

$$\lambda_{max} = (au + bv)^2 + (cu + dv)^2 \tag{4.49}$$

Let us now apply the method of Lagrange multipliers. From equations (4.49) and (4.47) we get:

$$\wedge(u, v, \zeta) = (au + bv)^2 + (cu + dv)^2 + \zeta(1 - u^2 - v^2) \tag{4.50}$$

Partially differentiating with respect to u, v and ζ then setting these equal to zero gives:

$$\frac{\partial \wedge (u, v, \zeta)}{\partial u} = 2(a^2 + c^2)u + 2(ab + cd)v - 2u\zeta = 0 \tag{4.51}$$

$$\frac{\partial \wedge (u, v, \zeta)}{\partial v} = 2(ab + cd)u + 2(b^2 + d^2)v - 2v\zeta = 0 \tag{4.52}$$

$$\frac{\partial \wedge (u, v, \zeta)}{\partial \zeta} = 1 - u^2 - v^2 = 0 \tag{4.53}$$

Dividing (4.51) and (4.52) by two, then re-arranging these results into matrix form we get:

$$\begin{bmatrix} a^2+c^2 & ab+cd \\ ab+cd & b^2+d^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \zeta \begin{bmatrix} u \\ v \end{bmatrix} \tag{4.54}$$

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \zeta \begin{bmatrix} u \\ v \end{bmatrix} \tag{4.55}$$

$$\mathbf{A}^T\mathbf{A} \begin{bmatrix} u \\ v \end{bmatrix} = \zeta \begin{bmatrix} u \\ v \end{bmatrix} \tag{4.56}$$

Therefore $\begin{bmatrix} u \\ v \end{bmatrix}$ is an eigenvector of $\mathbf{A}^T\mathbf{A}$ and ζ is an eigenvalue.

From equation (4.24) and (4.56) we can see the scaling factor for the solution $\begin{bmatrix} u \\ v \end{bmatrix}$ is $|\zeta|$.

For the transition function f to be a contraction mapping, the scaling factor λ must be $0 \leq \lambda < 1$, so both of the eigenvalues of $\mathbf{A}^T\mathbf{A}$ must be $0 \leq \zeta < 1$ and real.

If $\mathbf{A}$ consists only of integers, for its eigenvalues to meet these conditions $\mathbf{A}$ must be a lower-diagonal or upper-diagonal matrix. This is equivalent to saying that the calculation of the next state of each cell must depend upon just the state of the cell to its left or the state of the cell to its right. Note that CA that determine the next state of each cell according to the state of none of their neighbours would also converge, however the solution would be trivial and the properties of such a system are not of interest to us in our attempts to mimic morphogenesis.

Figure 4.10 shows the four one-dimensional two state additive CA that meet this criteria.

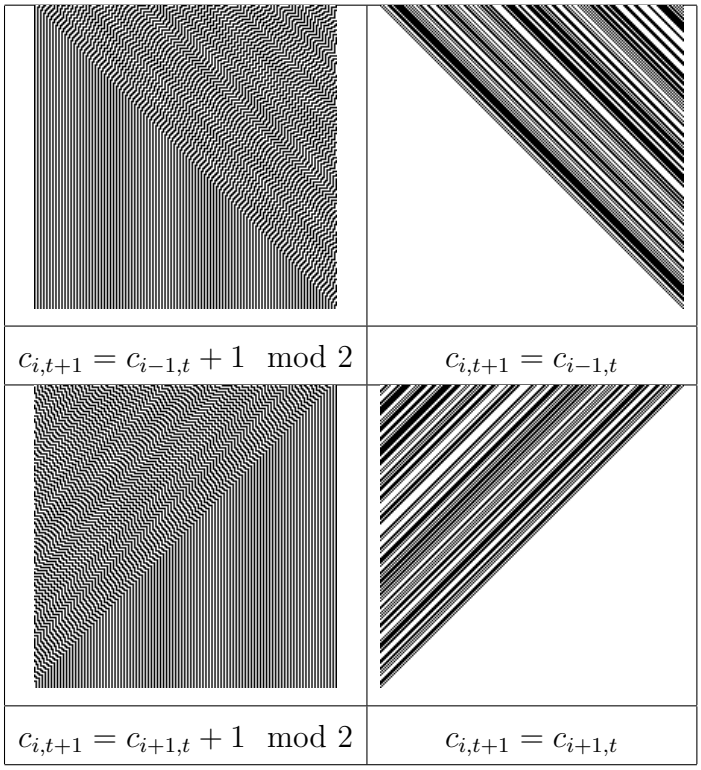


Figure 4.10: Convergent 1D 2 state additive CA

4.7 Two dimensions, from first to last state

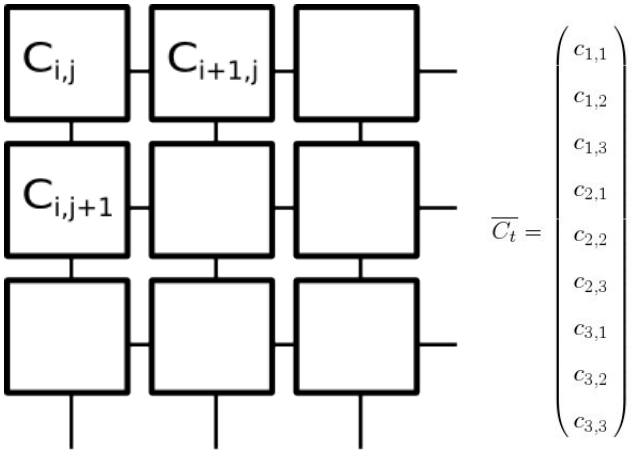
Let us index each cell of a two-dimensional CA with the tuple (i, j), then describe the state of each cell with an integer, $c_{i,j,t}$ and the pattern of the entire array as a matrix, $\mathbf{C}_t$ (see figure 4.11a).

If $\mathbf{C}_0$ is the initial pattern of cell states, $f(\mathbf{C}_0)$ is its subsequent pattern after one time step, and $f(f(\mathbf{C}_0))$ or $f^2(\mathbf{C}_0)$ is its pattern at $t = 2$; where f describes the transition from one iteration to the next. The matrix $\mathbf{C}_t$ is first transcribed into a row-major vector, $\overline{C_t}$ (figure 4.11b) in order for f to be a linear function of matrix algebra.

Let us now define a simple transition function, f for the next time step:

$$c_{i,j,t+1} = vc_{i,j-1,t} + wc_{i-1,j,t} + xc_{i+1,j,t} + yc_{i,j+1,t} + zc_{i,j,t} + d \tag{4.57}$$

where v, w, x, y and z are coefficients of the state of neighbours of each cell, and of the state of the cell itself; and d is a constant.



(a) (b)

Figure 4.11: Index of CA elements, and a row-major vector equivalent

A transition function for the entire array can be formed from (4.57) such that $f(C_t) = \mathbf{A}\overline{C_t} + \overline{D}$ where $\overline{D}$ is a constant and the transition matrix (for a 3 by 3 CA), $\mathbf{A}$, takes the form:

$$\mathbf{A} = \begin{pmatrix} z & x & 0 & y & 0 & 0 & 0 & 0 & 0 \\ w & z & x & 0 & y & 0 & 0 & 0 & 0 \\ 0 & w & z & 0 & 0 & y & 0 & 0 & 0 \\ v & 0 & 0 & z & x & 0 & y & 0 & 0 \\ 0 & v & 0 & w & z & x & 0 & y & 0 \\ 0 & 0 & v & 0 & w & z & 0 & 0 & y \\ 0 & 0 & 0 & v & 0 & 0 & z & x & 0 \\ 0 & 0 & 0 & 0 & v & 0 & w & z & x \\ 0 & 0 & 0 & 0 & 0 & v & 0 & w & z \end{pmatrix} \tag{4.58}$$

The spacing of the coefficients v, w, x, y and z within $\mathbf{A}$ depend on the size of the CA.

Because the form of f is the same as for the one-dimensional equivalent the analysis of the conditions for convergence is equally valid, as is the result — that $\mathbf{A}$ must be an upper-diagonal or lower-diagonal matrix. This analysis is also equally true of a CA of any number of dimensions.

Figure 4.12 shows the convergence of a two-dimensional two state CA that meets this criterion and uses the transition function:

$$c_{i,j,t+1} = c_{i-1,j,t} + c_{i,j_1,t} + 1 \mod 2 \tag{4.59}$$

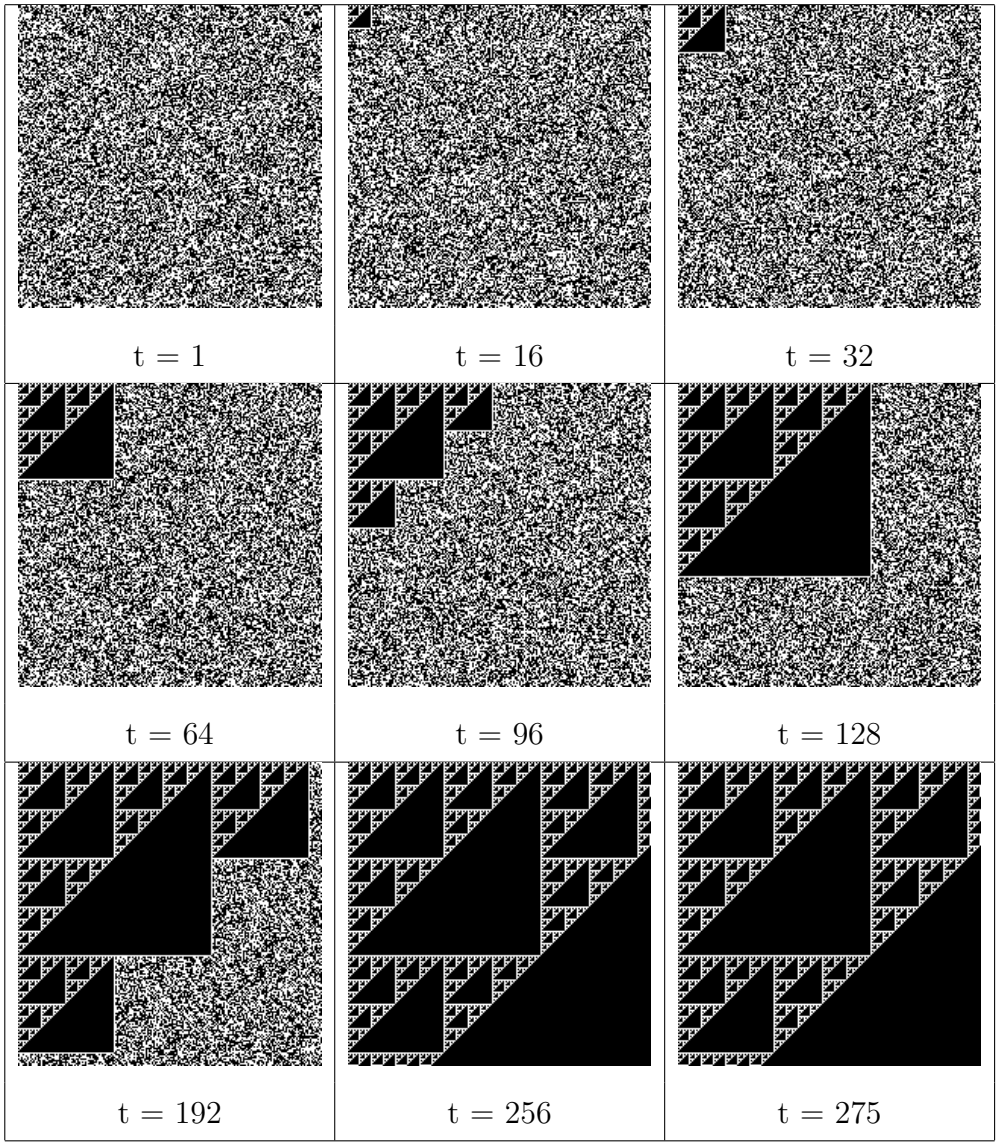


Figure 4.12: Convergent 2D 2 state CA

This bears some resemblance to the pattern found on the shell of the cone snail (see figure 4.13).

Note that the pattern of figure 4.13 appears from one corner and propagates towards its antipode. This can also be seen in Miller and Lui's self-assembling french flag

Figure 4.13: The cone snail

Image courtesy of Richard Ling

patterns of figures 3.4, 3.6, 3.7 and 3.8.

4.8 Conclusions

For an additive CA described by a transition function of integers to converge to a single state from any initial state, the transition matrix must be upper-diagonal or lower-diagonal. Thus each cell must use just one state-input from neighbouring cells per axis to determine its next state. Also it cannot use its current state when determining its next state.

Chapter 5

Designing cellular automata to converge to specific patterns

This chapter will present a means of designing the local transition rules of one- and two-dimensional additive CA in order that the CA converge to a chosen pattern. The limitations of this additive CA architecture will then be analysed using an NP-complete search. The effect of introducing diagonals to the look-up table, and increasing the number of dimensions of the CA will be similarly analysed.

5.1 1D cellular automata design

A 1D CA converges if the next state of every cell depends only on the state of the cell to the left or to the right of it. (See chapter 3). Thus the transition matrix, $\mathbf{A}$, is either an upper-triangular matrix or a lower-triangular matrix, and $\mathbf{A}^n = 0$ if n is greater than the length of the CA. Thus (5.1) (derived from (4.14)) is simplified to (5.2).

$$\overline{C}_{t+n} = \mathbf{A}^n \overline{C}_{t=0} + D \frac{\mathbf{I} - \mathbf{A}^n}{\mathbf{I} - \mathbf{A}} \mod r \tag{5.1}$$

$$\overline{C}_{t+n} = \overline{D}\,\frac{\mathbf{I}}{\mathbf{I}-\mathbf{A}}\;|n \geq \text{Size}(\overline{C}) \mod r \tag{5.2}$$

This is rearranged to form (5.3).

$$(\mathbf{I}-\mathbf{A})\,\overline{C}_{t+n} = \overline{D} \tag{5.3}$$

(5.3) is expanded to form (5.4). Note we've now set v to zero so the next-state for each cell depends on the state of the cell to the immediate left of itself.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -x & 1 & 0 & 0 & 0 & 0 \\ 0 & -x & 1 & 0 & 0 & 0 \\ 0 & 0 & -x & 1 & 0 & 0 \\ 0 & 0 & 0 & -x & 1 & 0 \\ 0 & 0 & 0 & 0 & -x & 1 \end{bmatrix} \begin{bmatrix} c_{1,t+n} \\ c_{2,t+n} \\ c_{3,t+n} \\ c_{4,t+n} \\ c_{5,t+n} \\ c_{6,t+n} \end{bmatrix} = \begin{bmatrix} d \\ d \\ d \\ d \\ d \\ d \end{bmatrix} \mod r \tag{5.4}$$

(5.4) is simplified to form a series of simultaneous equations (5.5).

$$c_{1,t+n} = d \tag{5.5}$$
$$-xc_{1,t+n} + c_{2,t+n} = d \tag{5.6}$$
$$-xc_{2,t+n} + c_{3,t+n} = d \tag{5.7}$$
$$-xc_{3,t+n} + c_{4,t+n} = d \tag{5.8}$$
$$-xc_{4,t+n} + c_{5,t+n} = d \tag{5.9}$$
$$-xc_{5,t+n} + c_{6,t+n} = d \tag{5.10}$$

Given a value for C_{t+n} (the desired convergent CA form), the simultaneous equations of (5.5) are formed. If a solution to these equations exist, the values of x and d describe the transition rule of the CA.

5.2 Designing a 1D CA from the intended final pattern

Let us design a simple six cell one-dimensional CA such that it converges to form the following pattern:

1	0	1	0	1	0

Figure 5.1: Desired 1D 6 cell CA pattern

The diagonal of the matrix of equation (5.5) is populated with the desired CA pattern, such that (5.5) becomes:

$$1 = d \tag{5.11}$$

$$-x = d \tag{5.12}$$

$$0.x + 1 = d \tag{5.13}$$

$$-x = d \tag{5.14}$$

$$0.x + 1 = d \tag{5.15}$$

$$-x = d \tag{5.16}$$

From (5.11) we can determine that $x = -1$ and $d = 1$.

Figure 5.2(a) displays a CA of this design developing from null initial conditions to the pattern of figure 5.1 in six cycles. Figure 5.2(b) displays a CA of this design developing from random initial conditions to the pattern of figure 5.1 in six cycles.

$t=0$	0	0	0	0	0	0
$t=1$	1	1	1	1	1	1
$t=2$	1	0	0	0	0	0
$t=3$	1	0	1	1	1	1
$t=4$	1	0	1	0	0	0
$t=5$	1	0	1	0	1	1
$t=6$	1	0	1	0	1	0
$t=7$	1	0	1	0	1	0
$t=8$	1	0	1	0	1	0

(a)

$t=0$	0	1	3	0	2	0
$t=1$	1	1	0	-2	1	-1
$t=2$	1	0	0	1	3	0
$t=3$	1	0	1	1	0	-2
$t=4$	1	0	1	0	0	1
$t=5$	1	0	1	0	1	1
$t=6$	1	0	1	0	1	0
$t=7$	1	0	1	0	1	0
$t=8$	1	0	1	0	1	0

(b)

Figure 5.2: 1D 6 cell CA developing from null and random initial conditions

5.3 Designing a 2D CA from the intended final pattern

A two-dimensional CA is robust if the next state of every cell depends only upon the state of two cells, one to the left or right of it and one above or below it (see Chapter 4). Therefore the transition matrix, $\mathbf{A}$, is either an upper-triangular matrix or a lower-triangular matrix, and $\mathbf{A}^n = 0$ if n is greater than the dimensions of C. A system of simultaneous equations are formed by setting $(\mathbf{I} - \mathbf{A})\,\overline{C}_{t+n}$ equal to $\mathbf{D}$, thus forming (5.17)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -x & 1 & 0 & 0 \\ -w & 0 & 1 & 0 \\ 0 & -w & -x & 1 \end{bmatrix} \begin{bmatrix} c_{11,t+n} \\ c_{12,t+n} \\ c_{21,t+n} \\ c_{22,t+n} \end{bmatrix} = \begin{bmatrix} d \\ d \\ d \\ d \end{bmatrix} \tag{5.17}$$

Given a value for $\overline{C}_{t+n}$ (the desired convergent CA pattern), the simultaneous equations of (5.17) are formed. The values of x, w and d describe the transition rule of

the CA if a solution to the equations exists.

5.4 2D cellular automata design example

Let us design a transition function for a two by two cell CA such that it converges to the following pattern:

1	0
0	1

Figure 5.3: Desired 2D 6 cell CA pattern

$$1 = d \tag{5.18}$$

$$-x = d \tag{5.19}$$

$$-w = d \tag{5.20}$$

$$-0.w + -0.x + 1 = d \tag{5.21}$$

From (5.18) we can determine that $x = -1$,$w = -1$ and $d = 1$.

Figure 5.4 displays a CA of this design developing from null initial conditions to the pattern of figure 5.3 in four cycles. Figure 5.5 displays a CA of this design developing from random initial conditions to the pattern of figure 5.3 in four cycles.

$t = 0$

0	0
0	0

$t = 1$

1	1
1	1

$t = 2$

1	0
0	-1

$t = 3$

1	0
0	1

Figure 5.4: 2D 4 cell CA developing from null initial conditions

$t = 0$

0	1
3	0

$t = 1$

1	1
1	-3

$t = 2$

1	0
0	0

$t = 3$

1	0
0	1

Figure 5.5: 2D 6 cell CA developing from random initial conditions

5.5 Limitations on possible cellular automata states

We have shown that every cell of a 1D convergent CA updates its state based on the value of one input, and that every cell of a 2D convergent CA updates its state based on the values of one input on each axis. In the event that the simultaneous linear equations formed by the two inputs and the output of each cell contradict, such a CA state will be impossible. Creating the simultaneous linear equations for figure 5.6 demonstrates this.

1	2
2	1

Figure 5.6: An impossible CA state

$$0.x + 0.w + d = 1 \tag{5.22}$$

$$1.x + 0.w + d = 2 \tag{5.23}$$

$$0.x + 1.w + d = 2 \tag{5.24}$$

$$2.x + 2.w + d = 1 \tag{5.25}$$

Equation (5.25) contradicts equations (5.24), (5.23) and (5.22).

5.6 Significance of additive CA limitations

Given the size of the CA, for instance a ten by ten CA, and the size of the alphabet as defined by the modulus, r, there are r^{100} different CA states.

Given the three variables, x, w and d, their limit defined by R (since this defines the size of the usable alphabet of the CA), the number of different rules and thus the number of possibly different CA states is r^3.

The final CA state is a function of the variables x, w, d, R and the size of the array. Some combinations of these variables result in the same final CA state.

Thus calculating the actual number of possible stable CA states is a non-trivial task. See listing 5.1 for the pseudo-code of an exhaustive search program. The results of this empirical analysis are shown in figures 5.7 and 5.8.

```
1 for alphabet = 0 to 12
2         for width,height of array = 1 to 20
3                 results = new array
4                 for x,w,k < alphabet
5                         for R < alphabet+1
6                                 create and iterate CA width+height times
7                                 if final CA pattern is not already in results array,
                                      add to array
8                 show length of results array
```

Listing 5.1: Exhaustive CA analysis

This shows that a next-state rule that is formed from a linear combination of neighbouring states will limit the number of possible patterns that can be formed by the CA.

5.7 Effect of increasing the dimensions of the CA

Figure 5.9 shows the neighbourhood function of a stable CA of sixteen cells in four-dimensions, represented as multiple two-dimension arrays. Equation (5.26) is its corresponding transition matrix. The column vector $\overline{C}_t$ is now a row-column major

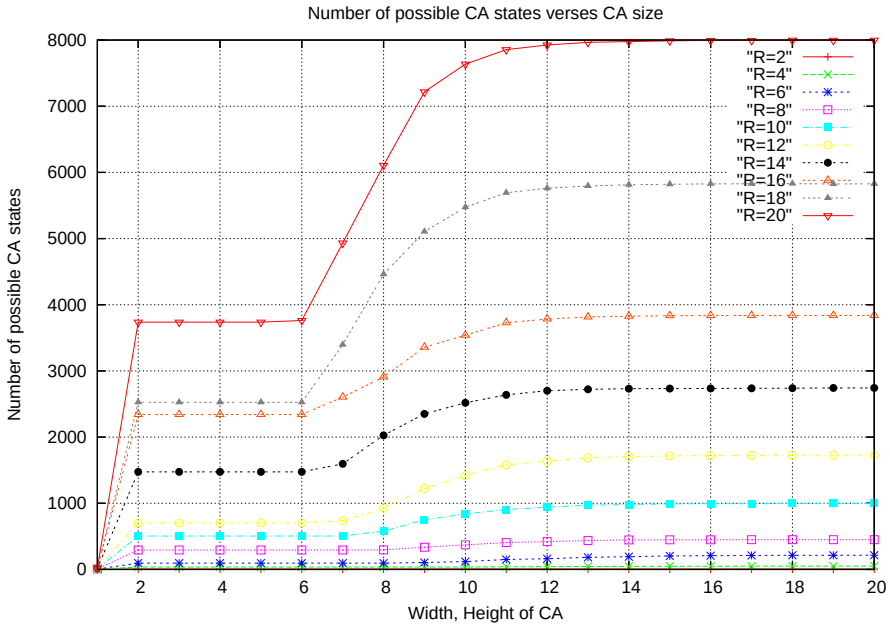


Figure 5.7: Number of different CA states verses CA size. R is the CA modulus.

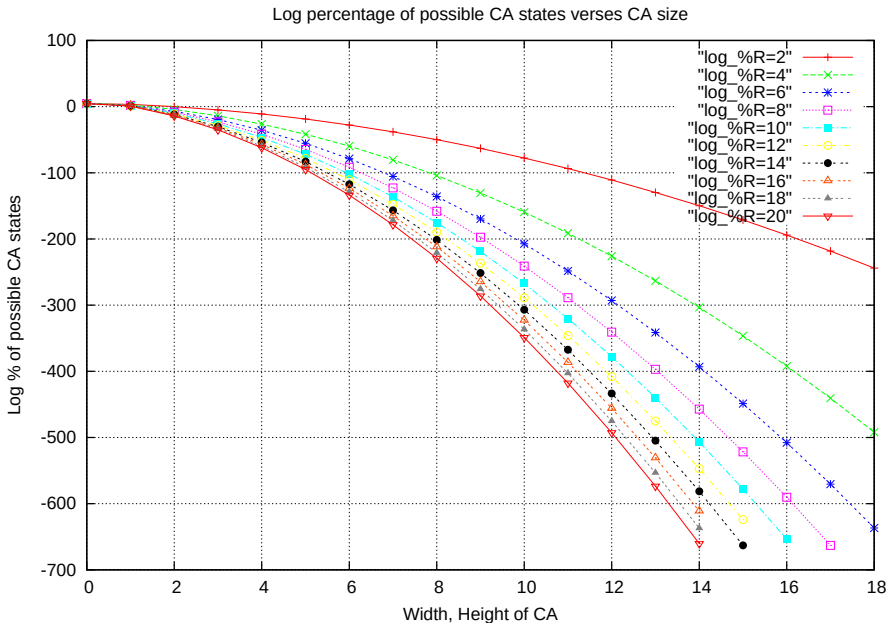


Figure 5.8: Log percentage of different CA states verses CA size. R is the CA modulus.

vector.

For a sixteen cell system in four dimensions:

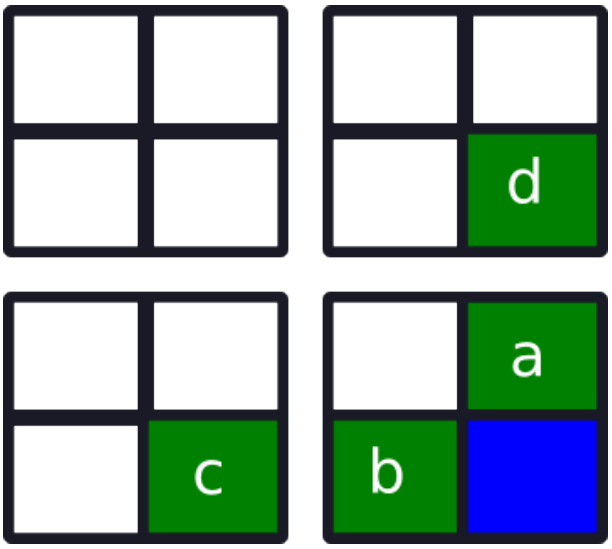


Figure 5.9: The neighbourhood function of a stable CA of 16 cells in 4-D

$$T = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
b & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & b & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & c & 0 & 0 & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & c & 0 & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & c & 0 & b & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & d & 0 & 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & d & 0 & 0 & 0 & 0 & 0 & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & d & 0 & 0 & 0 & 0 & 0 & b & a & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & d & 0 & 0 & 0 & c & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 & c & 0 & 0 & a & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 & c & 0 & b & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 0 & c & 0 & b & a & 0
\end{pmatrix} \tag{5.26}$$

The number of possible CA states, L depends upon the size of the CA alphabet, r, the width of the CA, x and the CA dimensions, d, according to equation (5.27).

$$L = r^{x^d} \tag{5.27}$$

The number of possible CA rules, and thus given a sufficiently large x the number of CA states it is possible to form as stable, M is given by equation (5.28).

$$M = r^{d+1} \tag{5.28}$$

Thus the percentage of possible CA states, N, that can be formed is given by equation (5.29). Figure 5.10 is a logarithmic plot of N versus the number of dimensions for increasing values of A.

$$N = 100.r^{d+1-x^d} \tag{5.29}$$

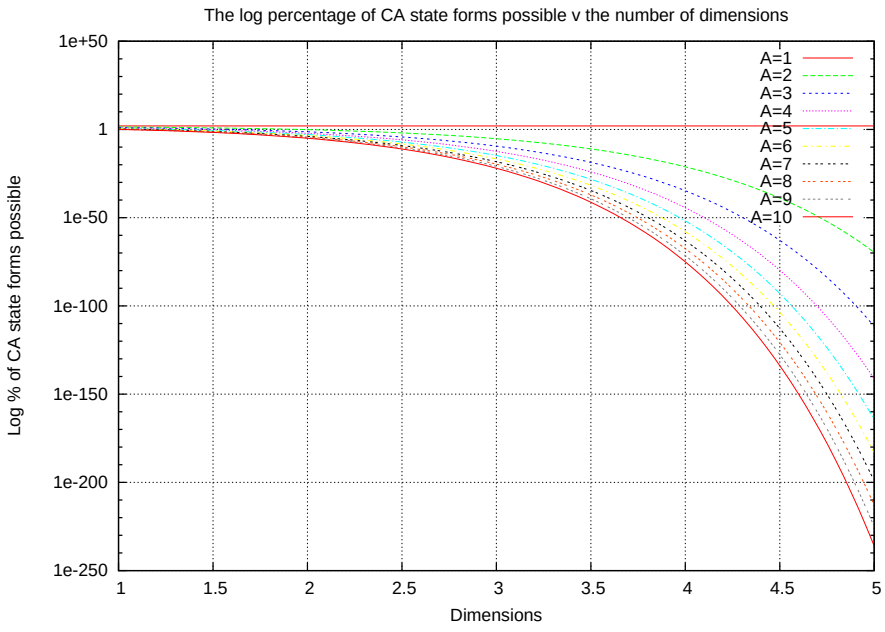


Figure 5.10: A logarithmic plot of N versus the number of dimensions

This shows that as the number of dimensions the CA exists in increase, the number of possible stable CA patterns formed by an additive transition rule also increase. However this increase is smaller than the increase in the number of impossible stable CA patterns.

5.8 Effect of using a Moore neighbourhood

If the neighbourhood function is expanded to include all cells that are adjacent to the cell, not just those orthogonally adjacent, the number of possible stable CA rules increases (provided the number of dimensions is greater than 1).

Figure 5.11 shows the augmented neighbourhood function of a stable CA of two dimensions. Equation (5.30) is the transition matrix for this CA. Note that the matrix is still lower-diagonal, thus the CA is stable.

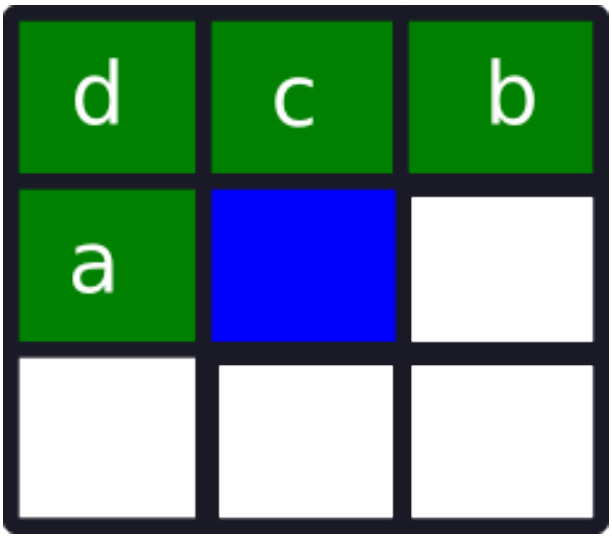


Figure 5.11: The augmented neighbourhood function of a stable CA of 2 dimensions

$$T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ c & b & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ d & c & b & a & 0 & 0 & 0 & 0 & 0 \\ 0 & d & c & 0 & a & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & b & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d & c & b & a & 0 & 0 \\ 0 & 0 & 0 & 0 & d & c & 0 & a & 0 \end{pmatrix} \tag{5.30}$$

Figure 5.12 shows the augmented neighbourhood function of a stable CA of three dimensions.

Figure 5.12: The augmented neighbourhood function of a stable CA of 3 dimensions

Thus it can be seen that the size of the augmented neighbourhood, O varies with d according to equation (5.31), and the percentage of possible CA states, N, that can be formed is given by equation (5.32).

$$O = \frac{3^d - 1}{2} \tag{5.31}$$

$$N = r^{\frac{3^d-1}{2} - x^d} \tag{5.32}$$

Figure 5.13 is a logarithmic plot of N versus the number of dimensions for both the orthogonal and augmented neighbourhood functions.

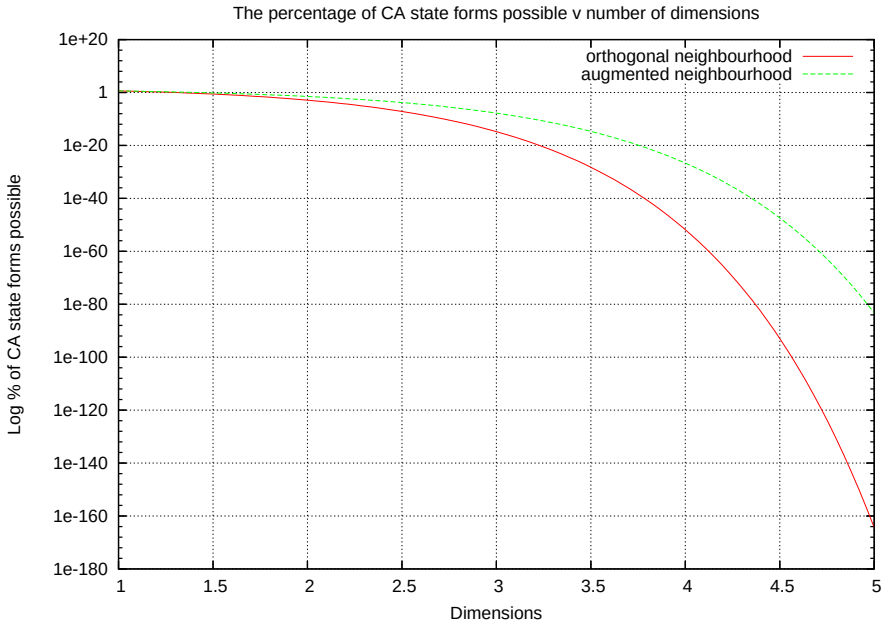


Figure 5.13: A logarithmic plot of N versus the number of dimensions for both the orthogonal and augmented neighbourhood functions

Figure 5.14 is a logarithmic plot of N versus the size of the alphabet for both the orthogonal and augmented neighbourhood functions.

5.9 Conclusions

In this chapter we have demonstrated the process of designing CA to converge to specific patterns using an additive transition function and simultaneous equations. Since there are a greater number of simultaneous equations than there are coefficients to solve them for, there sometimes exist CA patterns that cannot be converged upon using additive CA. Using a brute-force analysis we have shown this technique is limited to creating a small percentage of all possible CA patterns as stable final states

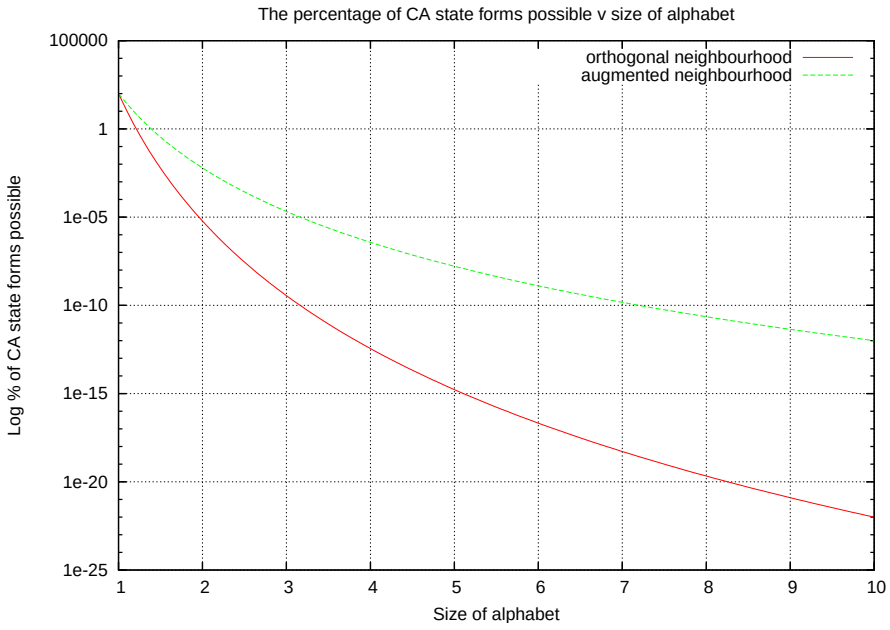


Figure 5.14: A logarithmic plot of N versus the size of the alphabet for both the orthogonal and augmented neighbourhood functions

Chapter 6

Non-linear cellular automata design

Since there are may be more cells in a CA than there are coefficients of neighbouring cells, every possible CA transition function cannot be represented by the additive CA model shown in the previous chapter. An alternative to this is to introduce non-linear components to the transition function.

For instance, referring to example 5.6 from the previous chapter, a solution is to introduce a $c_{t,i-1,j}.c_{t,i,j-1}$ term to the transition function:

$$c_{t+1,i,j} = xc_{t,i-1,j} + wc_{t,i,j-1} + yc_{t,i-1,j}c_{t,i,j-1} + d \tag{6.1}$$

Four simultaneous equations with four variables can be derived from (6.1).

$$0.x + 0.w + 0.0.y + d = 1 \quad (6.2)$$

$$1.x + 0.w + 1.0.y + d = 2 \quad (6.3)$$

$$0.x + 1.w + 0.1.y + d = 2 \quad (6.4)$$

$$2.x + 2.w + 2.2.y + d = 1 \quad (6.5)$$

Thus $w = 1$, $x = 1$, $y = -1$ and $d = 1$.

A more general solution is to represent the transition function as a look-up table, the inputs of which would be the present state of the neighbouring cells, the output the next state of the cell. This chapter will prove that such a design is also constrained to a one-input-per-axis design, then present a design algorithm for the formation of the look-up table.

6.1 A sum-of-products representation of look-up tables

Any system of combinatorial logic can be represented by a look-up table (LUT). Figure 6.1(a) shows the LUT for a two-input XOR gate. Any LUT can be represented by a sum-of-products (SOP) boolean expression. If each row of the LUT for which the output is high is represented as a boolean product, then a boolean sum of each product forms the complete expression. Figure 6.1(b) shows the SOP of a two-input XOR gate.

Thus the SOP representation of an XOR gate is $f(a, b) = \overline{a}b + \overline{b}a \mod 2$.

An SOP representation of any two-input boolean product is $(a + k_1)(b + k_2) \mod 2$ where $k_1, k_2 \in \{0, 1\}$. Note that this is commutable and distributable, so also equal to $ab + k_2a + k_1b + k_1k_2 \mod 2$. A general case, g, for n-input boolean products and boolean sum of products is:

a	b	f(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

(a)

Boolean products
0
$\overline{a}b$
$a\overline{b}$
0

(b)

Figure 6.1: An XOR gate look-up table and its sum-of-products expression

$$(c_1 + k_1)(c_2 + k_2) \cdots (c_n + k_n) = \prod_i (c_i + k_i) \mod 2 \tag{6.6}$$

$$g = \sum_j \prod_i (c_i + k_{i,j}) \mod 2 \tag{6.7}$$

If we use a LUT to determine the next state of each cell within a two state six cell one-dimensional CA, the transition function would look like:

$$
\begin{bmatrix} c_{1,t+1} \\ c_{2,t+1} \\ c_{3,t+1} \\ c_{4,t+1} \\ c_{5,t+1} \\ c_{6,t+1} \end{bmatrix} = \sum_i \quad \left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ v_i & 0 & 0 & 0 & 0 & 0 \\ 0 & v_i & 0 & 0 & 0 & 0 \\ 0 & 0 & v_i & 0 & 0 & 0 \\ 0 & 0 & 0 & v_i & 0 & 0 \\ 0 & 0 & 0 & 0 & v_i & 0 \end{bmatrix} \begin{bmatrix} c_{1,t} \\ c_{2,t} \\ c_{3,t} \\ c_{4,t} \\ c_{5,t} \\ c_{6,t} \end{bmatrix} + \begin{bmatrix} k_{i,1} \\ k_{i,1} \\ k_{i,1} \\ k_{i,1} \\ k_{i,1} \\ k_{i,1} \end{bmatrix} \right)
$$

$$
\times \left(\begin{bmatrix} w_i & 0 & 0 & 0 & 0 & 0 \\ 0 & w_i & 0 & 0 & 0 & 0 \\ 0 & 0 & w_i & 0 & 0 & 0 \\ 0 & 0 & 0 & w_i & 0 & 0 \\ 0 & 0 & 0 & 0 & w_i & 0 \\ 0 & 0 & 0 & 0 & 0 & w_i \end{bmatrix} \begin{bmatrix} c_{1,t} \\ c_{2,t} \\ c_{3,t} \\ c_{4,t} \\ c_{5,t} \\ c_{6,t} \end{bmatrix} + \begin{bmatrix} k_{i,2} \\ k_{i,2} \\ k_{i,2} \\ k_{i,2} \\ k_{i,2} \\ k_{i,2} \end{bmatrix} \right)
$$

$$
\times \left(\begin{bmatrix} 0 & x_i & 0 & 0 & 0 & 0 \\ 0 & 0 & x_i & 0 & 0 & 0 \\ 0 & 0 & 0 & x_i & 0 & 0 \\ 0 & 0 & 0 & 0 & x_i & 0 \\ 0 & 0 & 0 & 0 & 0 & x_i \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_{1,t} \\ c_{2,t} \\ c_{3,t} \\ c_{4,t} \\ c_{5,t} \\ c_{6,t} \end{bmatrix} + \begin{bmatrix} k_{i,3} \\ k_{i,3} \\ k_{i,3} \\ k_{i,3} \\ k_{i,3} \\ k_{i,3} \end{bmatrix} \right) \mod 2
$$

$$
= \sum_i \quad (\mathbf{V_i}\overline{C_t} + \overline{K}_{i,1})(\mathbf{W_i}\overline{C_t} + \overline{K}_{i,2})(\mathbf{X_i}\overline{C_t} + \overline{K}_{i,3}) \mod 2
$$

$$
\begin{aligned}
= \sum_i \quad & (\mathbf{V_i}\mathbf{W_i}\mathbf{X_i}\overline{C_t}^3 + \mathbf{V_i}\mathbf{X_i}\overline{K_{i,2}C_t}^2 + \mathbf{W_i}\mathbf{X_i}\overline{K_{i,1}C_t}^2 + \overline{K_{i,1}K_{i,2}}\mathbf{X_i} \\
& + \mathbf{V_i}\mathbf{W_i}\overline{K_{i,3}C_t}^2 + \mathbf{V_i}\overline{K_{i,2}K_{i,3}C_t} + \mathbf{W}\overline{K_{i,1}K_{i,3}C_t} + \overline{K_{i,1}K_{i,2}K_{i,3}}) \mod 2
\end{aligned}
$$

Let us now consider the constraints on $\mathbf{V_i}, \mathbf{W_i}, \mathbf{X_i}$ and $\mathbf{K_{i,j}}$ such that the CA will always converge.

6.2 General case convergence analysis of cellular automata

For the purposes of this analysis we are only interested in the independence of $\overline{C}_n$ from $\overline{C}_0$, so we can ignore the powers of $\overline{C}_t$ and combine the coefficients into an all-encompassing matrix $\mathbf{A}_i$. We are interested in the form $\mathbf{A}_i$ must take.

$$g(\overline{C_0}) = \sum_{i=0}^{k} (\mathbf{A}_i \overline{C_0} + \overline{K_i}) \quad \text{where } \mathbf{A_i} = \begin{bmatrix} w_i & x_i & 0 & 0 & 0 & 0 \\ v_i & w_i & x_i & 0 & 0 & 0 \\ 0 & v_i & w_i & x_i & 0 & 0 \\ 0 & 0 & v_i & w_i & x_i & 0 \\ 0 & 0 & 0 & v_i & w_i & x_i \\ 0 & 0 & 0 & 0 & v_i & w_i \end{bmatrix} \tag{6.8}$$

Expanding $g^n(C_0)$, when k is greater than three, gives a coefficient of C_0 formed by the multinomial of the transition matrices:

$$(\mathbf{A}_0 + \mathbf{A}_1 + \mathbf{A}_2 \cdots + \mathbf{A}_m)^n \overline{C_0} \tag{6.9}$$

The multinomial expansion can be described as:

$$(x_1 + x_2 + \cdots + x_m)^n = \sum\nolimits_{k_0, k_1 \cdots, k_m} \binom{n}{k_0, k_1, \cdots, k_m} x_1^{k_0} x_2^{k_1} \cdots x_m^{k_m} \tag{6.10}$$

where the summation is taken over all sequences of $k_1, k_2, \cdots, k_m$ such that:

$$\sum_{i=1}^{m} k_i = n \tag{6.11}$$

and the multinomial coefficient can be expressed as:

$$\binom{n}{k_0, k_1, \cdots, k_m} = \frac{n!}{k_0!k_1!\cdots k_m!} \tag{6.12}$$

Thus the coefficients of every $\overline{C_0}$ term are constructed from a multinomial coefficient and n members of the set $\mathbf{A}$ of transition matrices:

$$\mathbf{A} = \{\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2, \cdots, \mathbf{A}_k\} \tag{6.13}$$

For every coefficient of $\overline{C_0}$ to be zero, every possible product of n members of $\mathbf{A}$ must be zero. Thus the convergence criteria for f also holds true for g: every member of $\mathbf{A}$ must be a lower-diagonal matrix or every member must be an upper-diagonal matrix. For this to be so, x_i must equal zero. Also either v or x must equal zero. That is, the same criteria for an additive CA to converge also applies to any CA that can describe its transition function with a look-up table; there can only be one input per axis and it cannot include inputs from its own current state.

6.3 Design of a look-up table transition functions

Now we know the design-constraints for a convergent CA we can begin to design them. Figure 6.2 shows a hardware implementation of a cell that could form part of a synchronous CA. The next-state arithmetic is performed by the logic block $f()$ and the state is stored in a local register and shared with its neighbours.

The LUT determines the next state of each cell, and each cell uses an identical LUT to do so. The entries in this LUT, the combinatorial logic it represents, will be responsible for the final state of the convergent CA.

From chapter 3 we know that if the combinatorial logic function g represented by the transition function is to result in a convergent sequence within the CA metric space, after sufficient iterations, the output must remain constant.

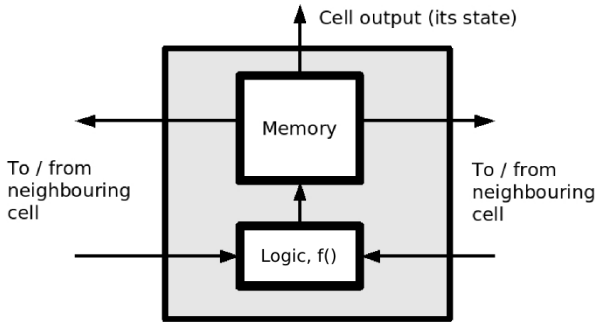


Figure 6.2: An implementation of a LUT-based cell

1	2	3
1	2	3
1	2	3

Figure 6.3: A 9 cell French flag pattern

$$\overline{C}_{n+1} = \overline{C_n} \mid n > \text{width+height of CA} \tag{6.14}$$

$$= g(\overline{C_n}) \tag{6.15}$$

Thus we can begin populating the LUT entries. At every time-step, including those after $t > n$, each cell will use its inputs to determine its next state. Given that the CA converges correctly, we know what those inputs will be from the final pattern. For instance, consider the simple French flag pattern shown in figure 6.3 where 1=red, 2=white, 3=blue.

Each cell will use two inputs to determine its next state. Let us assume they are from above and to the left of itself. If the boundaries of the CA are set to zero, the rules each cell must obey can be seen in figure 6.4.

The remaining LUT entries may have roles to play during the convergence of the CA, but their values are not critical to the formation of the correct pattern. Therefore we can set them to equal zero. Listing 6.1 shows the pseudo-code for the design of g.

```
1 Create a LUT entry for every possible combination of two-inputs
```

$g(0,0)=1$	$g(0,1)=2$	$g(0,2)=3$
$g(1,0)=1$	$g(2,1)=2$	$g(3,2)=3$
$g(1,0)=1$	$g(2,1)=2$	$g(3,2)=3$

above	left	output
0	0	1
0	1	2
0	2	3
1	0	1
2	1	2
3	2	3

Figure 6.4: The rules a 9 cell French flag CA must obey

```
2 Assign each entry the output = 0
3 For each cell in the CA pattern:
4         Determine the inputs to the cell assuming it has successfully converged. The
              inputs come from above and to the left of itself.
5         Find the LUT entry that corresponds to this combination of two-inputs, and re-
              assing its output to that of the cell state.
```

Listing 6.1: Design of LUT CA pseudo-code

Figure 6.5 displays a CA of this design developing from null initial conditions to the French-flag pattern in six cycles. Figure 6.6 displays a CA of this design developing from random initial conditions to the French-flag pattern in six cycles.

$t=0$

0	0	0
0	0	0
0	0	0

$t=1$

1	1	1
1	1	1
1	1	1

$t=2$

1	2	2
1	0	0
1	0	0

$t=3$

1	2	3
1	2	0
1	2	1

$t=4$

1	2	3
1	2	3
1	2	3

$t=5$

1	2	3
1	2	3
1	2	3

Figure 6.5: Development of a 9 cell CA from null conditions to a French flag pattern

There still exist certain patterns that cannot be formed by convergent CA based on

$t = 0$

1	1	3
0	1	2
0	2	0

$t = 3$

1	2	3
1	2	3
1	2	3

$t = 1$

1	2	2
1	1	3
1	1	0

$t = 4$

1	2	3
1	2	3
1	2	3

$t = 2$

1	2	3
1	2	3
1	2	3

$t = 5$

1	2	3
1	2	3
1	2	3

Figure 6.6: Development of a 9 cell CA from random conditions to a French flag pattern

transition functions of the form f or the form g. This is because the rules required by each cell to form the final pattern may contradict each other. For instance, let us consider a six by six French flag (figure 6.7).

1	1	2	2	3	3
1	1	2	2	3	3
1	1	2	2	3	3
1	1	2	2	3	3
1	1	2	2	3	3
1	1	2	2	3	3

Figure 6.7: A 6 by 6 CA pattern that cannot be formed by g

The rule formed for cell $c_{1,2}$ (indexed $c_{x,y}$ from 1 to 6) is $g(0,1) = 1$, however the rule for cell $c_{1,3}$ is $g(0,1) = 2$. Clearly g cannot give two different outputs for the same two-input combination. Other contradictions also exist in figure 6.7 - the cells of column 2 form rules that contradict those of column 3, the cells of column 4 form rules that contradict those of column 5.

The percentage of all CA states that are possible with a CA using boolean maths, depends upon the size of the CA and its alphabet. Figure 6.8 demonstrates this dependence for an eight cell one-dimensional CA by showing how the percentage of possible, stable, CA patterns varies with the size of its alphabet.

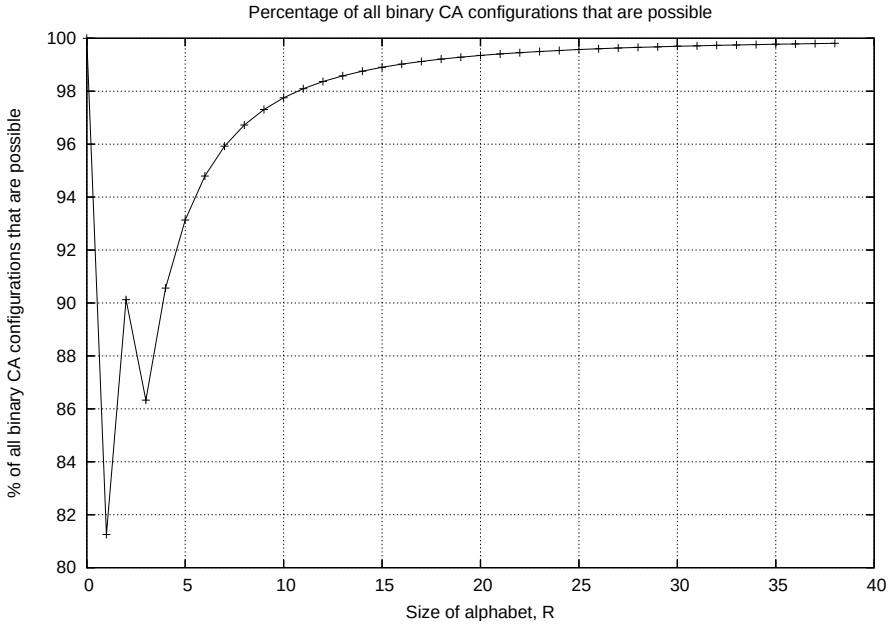


Figure 6.8: Percentage of all CA states that are possible using g, versus R

Thus if we increase the size of the alphabet available to each cell by adding states that don't appear in the output we can increase the probability of a design being possible. These extra states that are unused as outputs are "redundant states".

6.4 State redundancy

If a LUT-solution doesn't exist for a CA pattern then we can change the pattern to one for which a solution does exist. Then we can map the new pattern to the desired convergent pattern with another LUT. Figure 6.9 shows a possible hardware implementation of this approach.

There are some restrictions on which intermediate pattern the CA converges to. It must be possible to generate the LUT rules to generate the new pattern. Also the

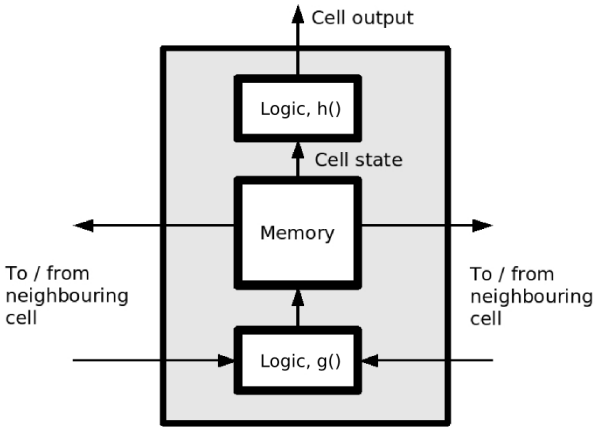


Figure 6.9: An implementation of a LUT-based cell with an alias output

same intermediate state cannot map to two or more different output states.

An extreme application of this approach is to assign each cell its own unique intermediate state (e.g. a co-ordinate reference) then store a complete co-ordinate to output mapping in $h()$, the logic unit responsible for mapping state to output. However, the more the states the larger the cell memory needs to be and the more entries are needed in each of the LUTs. Thus an optimum solution using as few states as possible needs to be found.

The algorithm shown in figure 6.10 achieves this by designing the CA LUTs in the order they converge to their correct state. If each cell takes its inputs from above and to the left of itself, the CA converges from the top-left corner. If each cell takes its inputs from below and to the right of itself, the CA converges from the bottom-right corner. Likewise for the top-right and bottom-left corners.

If the desired pattern is asymmetric, a more-optimal solution may exist if the CA converges left rather than right, up rather than down. Thus the algorithm designs a solution for all four input combinations then chooses the best.

Each cell state assignment is tested against two criteria:

1. This cell state acts as the input to two cells for which a state has already been assigned. Does the input-output combinations these form contradict existing

LUT entries for f?

2. Does the state-to-output mapping LUT entry for g contradict existing LUT entries?

If the answer to both these questions is no, the cell is assigned this state. If not, the cell is assigned another state. The algorithm first attempts to assign each cell a state that has been previously used in the CA design. If this, and other previously assigned states don't meet the two criteria, a new state is added to the CA alphabet and then assigned to this cell.

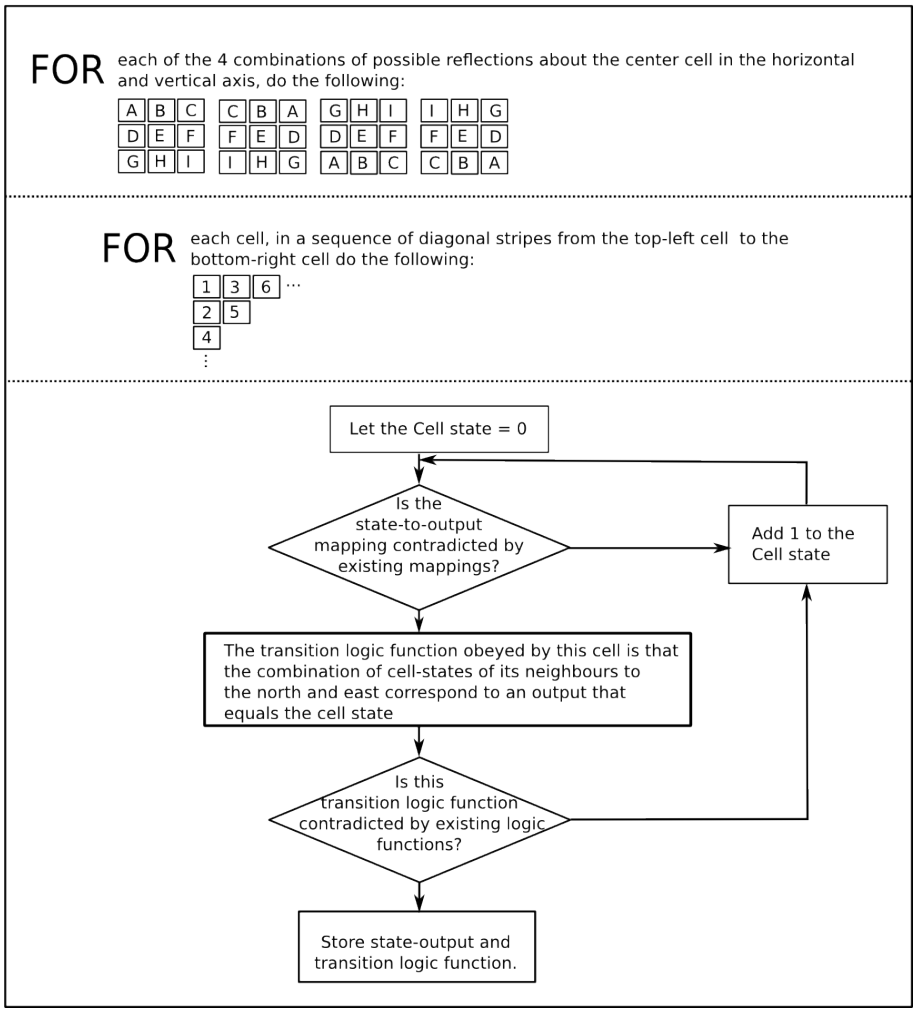


Figure 6.10: Design algorithm for g and $h()$

The source code for this algorithm can be found in appendix A.2.

As the number of possible CA states increases, so does the number of possible CA output patterns (see Figures 6.11 and 6.12).

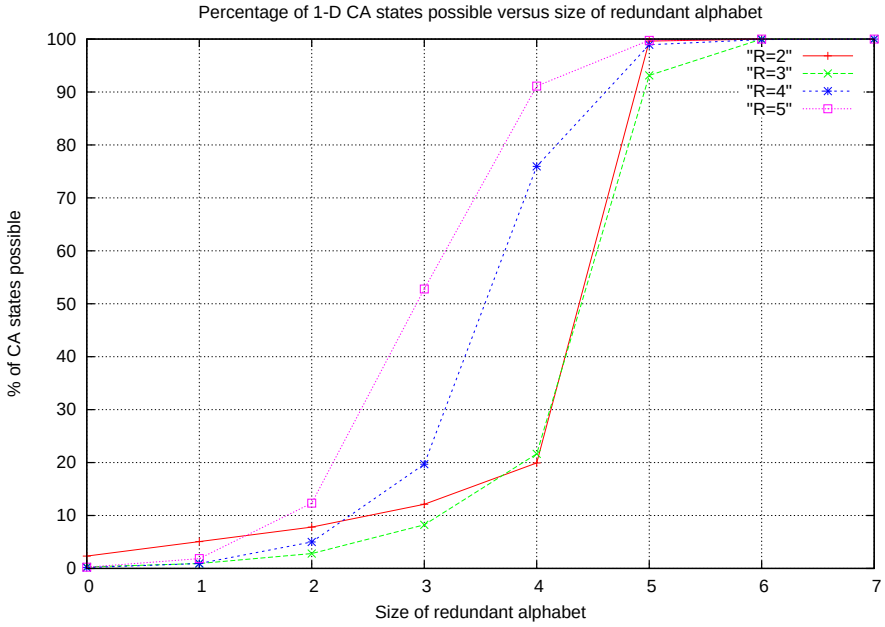


Figure 6.11: Percentage of all possible 1D CA states versus the size of the Redundancy alphabet. $R =$ Number of output states

6.5 Conclusions

Two new CA transition function schemes have been demonstrated. The first, a single LUT implementation of g, goes some way to removing the limitations on CA pattern imposed by the additive CA design, f. However there still exists certain patterns that cannot be converged upon using g alone. Thus a two-LUT scheme that uses an intermediate state and a state-to-output mapping has been introduced. This scheme is capable of creating a CA that will converge to any desired pattern. However, this comes at a cost to increased complexity of the cell transition function. Both schemes have been brute-force analysed to assess what percentage of possible patterns they can converge to, and the latter scheme has been analysed to find the

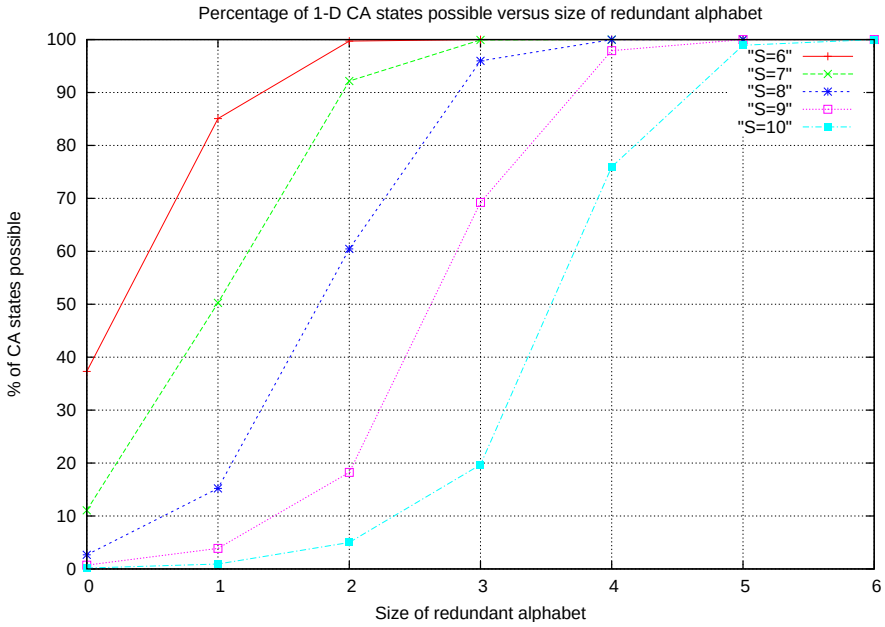


Figure 6.12: Percentage of all possible 1D CA states versus the size of the Redundancy alphabet $S =$ Size of CA

trade-off between this percentage and the increased complexity of the cell.

Chapter 7

Demonstrating robust patterns

This section will show solutions to Wolpert's French flag, a checkered pattern and the Welsh flag; designed using the algorithms presented in the previous chapters.

7.1 Developing a three-by-three French flag

This small design does not require redundancy to be robust, nor does it require an LUT solution; instead an algebraic solution (see equation (7.1)) suffices.

$$c_{t+1,x,y} = c_{t,x,y-1} + 1 \bmod 4 \tag{7.1}$$

Diagram 7.1 shows the development of the three-by-three French flag from an initial state of zero.

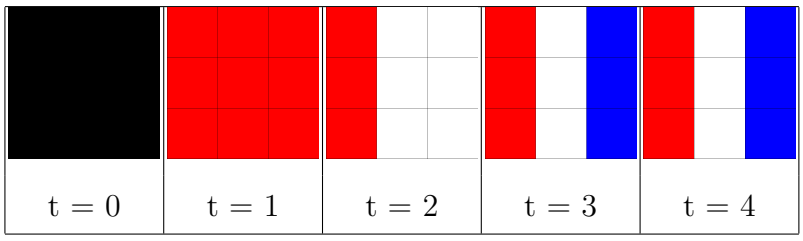


Figure 7.1: The development of a three-by-three French flag from the null state

Diagram 7.2 shows the development of the same three-by-three French flag from a corrupt starting state that has been formed with a random number generator.

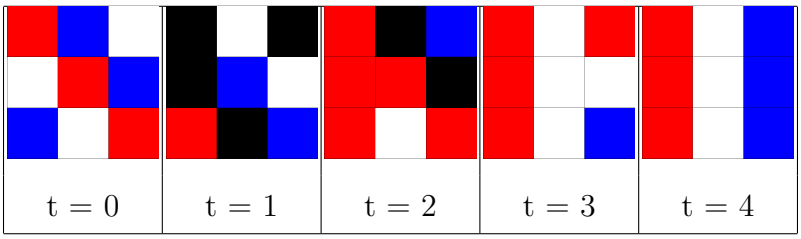


Figure 7.2: The development of a three-by-three French flag from a corrupt state

Figure 7.3 is a graph of the number of cells with incorrect states during the assembly cycle versus time, starting from null and corrupt initial conditions.

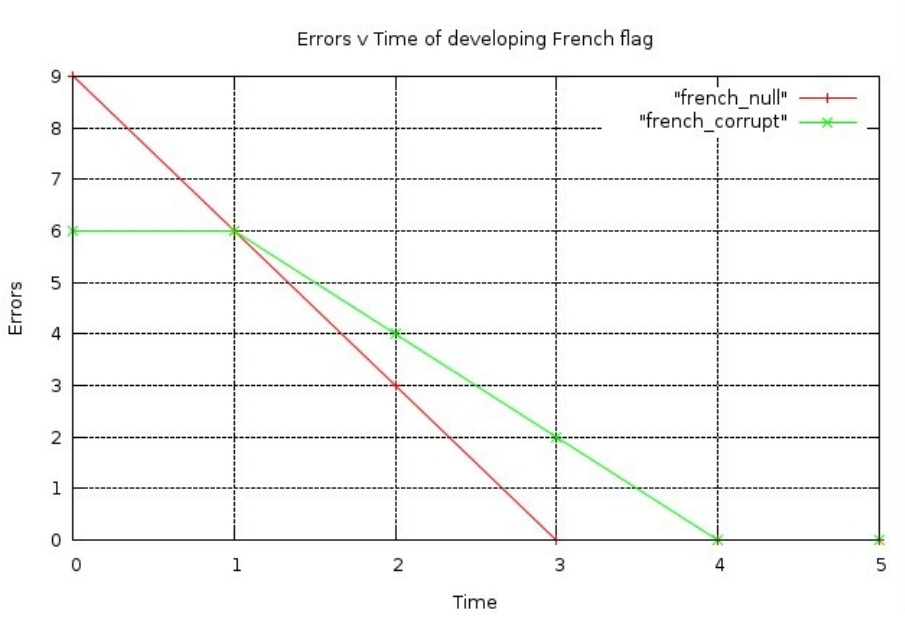


Figure 7.3: Errors v Time of developing French flag

7.2 Developing a twelve-by-twelve French flag

This design requires redundancy. Equation (7.2) is the cell transition function.

$$c_{t+1,x,y} = c_{t,x,y-1} + 1 \bmod 12 \tag{7.2}$$

The state to output mapping can be seen in figure 7.4.

State	Output
0 - 3	Red
4 - 7	White
8 - 11	Blue

Figure 7.4: The output rule of a twelve-by-twelve French flag

Diagram 7.5 shows the development of the twelve-by-twelve French flag from an initial state of zero.

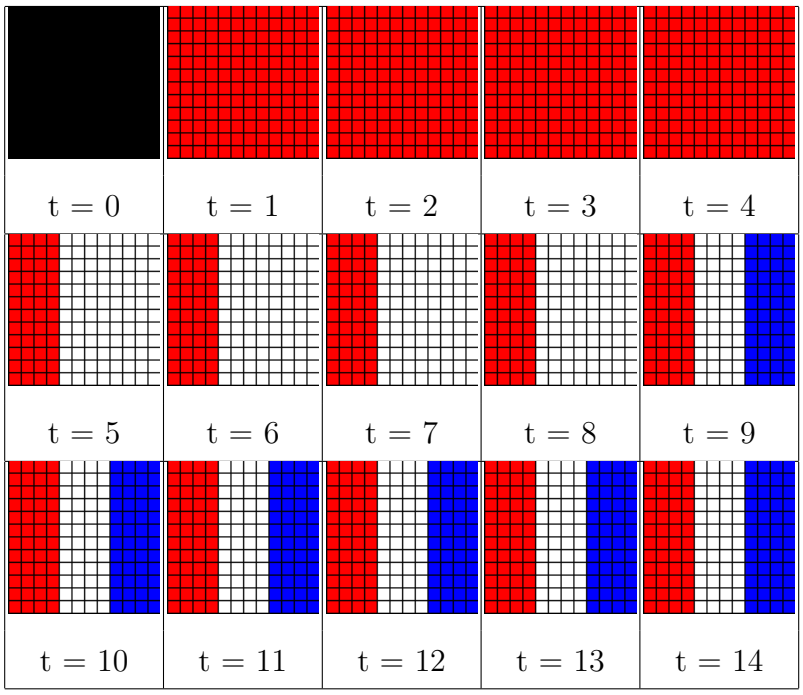


Figure 7.5: The development of a twelve-by-twelve French flag from the null state

Diagram 7.6 shows the development of the same twelve-by-twelve French flag from a corrupt starting state that has been formed with a random number generator.

Figure 7.7 is a graph of the number of cells with incorrect states during the assembly cycle versus time, starting from null and corrupt initial conditions.

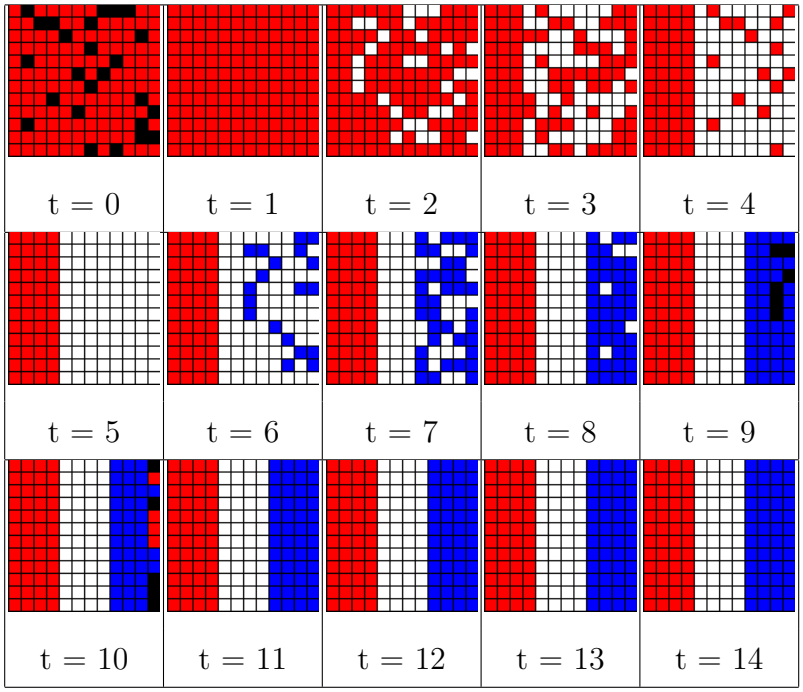


Figure 7.6: The development of a twelve-by-twelve French flag from a corrupt state

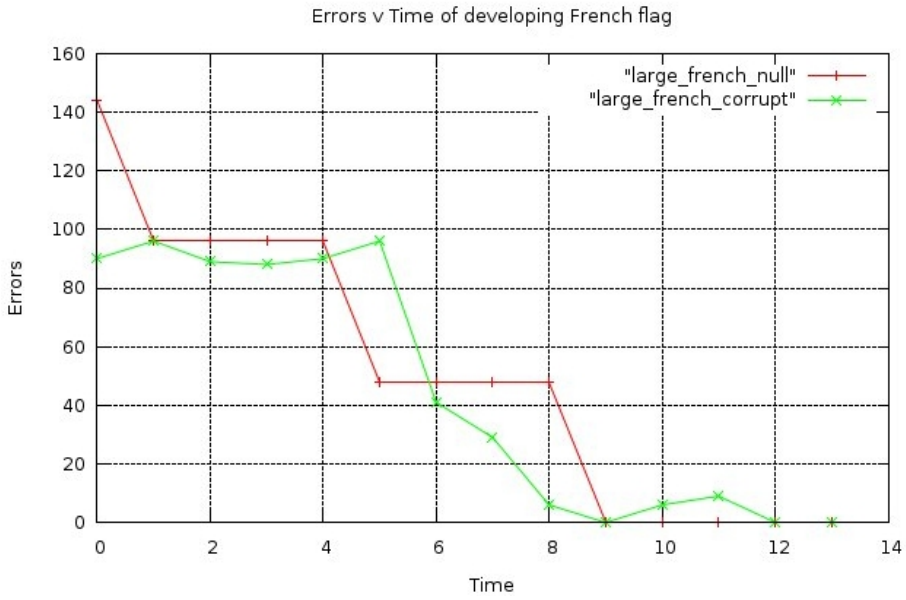


Figure 7.7: Errors v Time of developing French flag

7.3 Developing a sixteen-by-sixteen checkered pattern

A checkered pattern is formed with two redundant letters if diagonals are incorporated in the neighbourhood function. The next state LUT is derived from the following segment of CA form 7.8(a), with the remaining combinations corresponding to a next state of zero. The LUT of 7.8(b) shows the state to output mapping.

0	1	2	3	0
1	0	3	2	1
2	3	0	1	2
3	2	1	0	3
0	1	2	3	0

(a)

0 - 1	Red
2 - 3	White

(b)

Figure 7.8: A segment of the checkered pattern CA state map and its corresponding state-output mapping

Diagram 7.9 shows the development of the sixteen-by-sixteen checkered pattern from an initial state of zero.

Diagram 7.10 shows the development of the same sixteen-by-sixteen checkers flag from a corrupt starting state that has been formed with a random number generator.

Figure 7.11 is a graph of the number of cells with incorrect states during the assembly cycle versus time, starting from null and corrupt initial conditions.

7.4 Developing a 32 by 32 Welsh flag

The Welsh flag has been chosen, in part, for its complexity. The design requires 100 redundant letters.

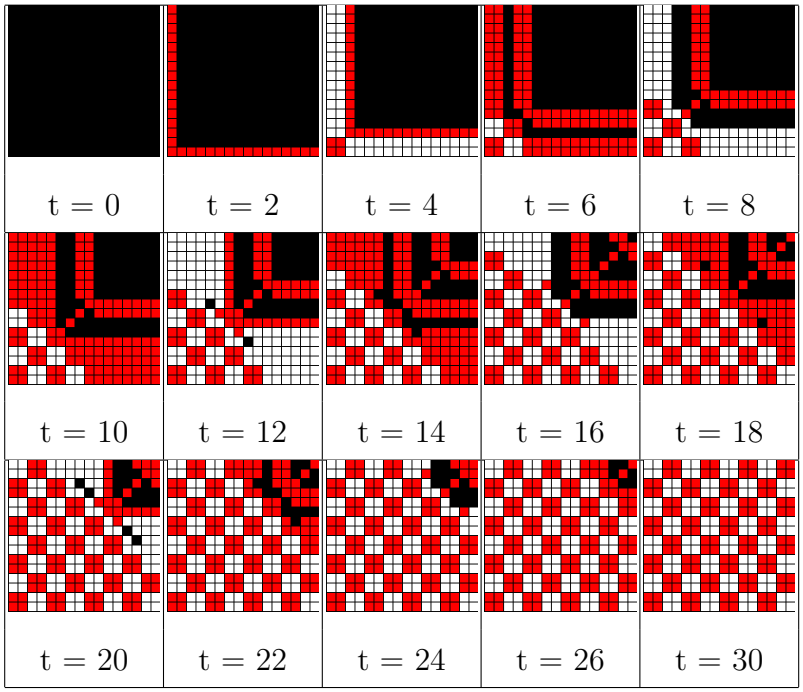


Figure 7.9: The development of a sixteen-by-sixteen checkered pattern from the null state

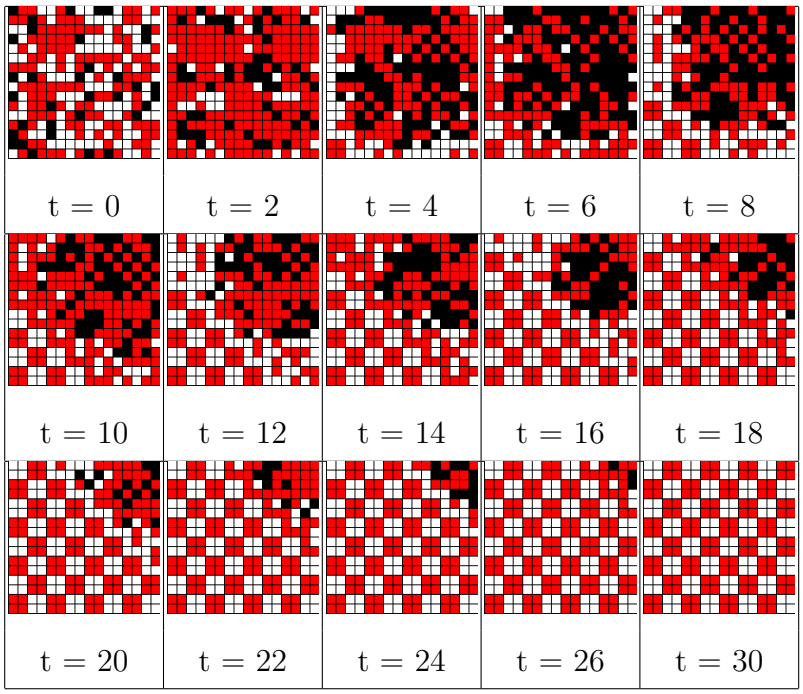


Figure 7.10: The development of a sixteen-by-sixteen checkers flag from a corrupt state

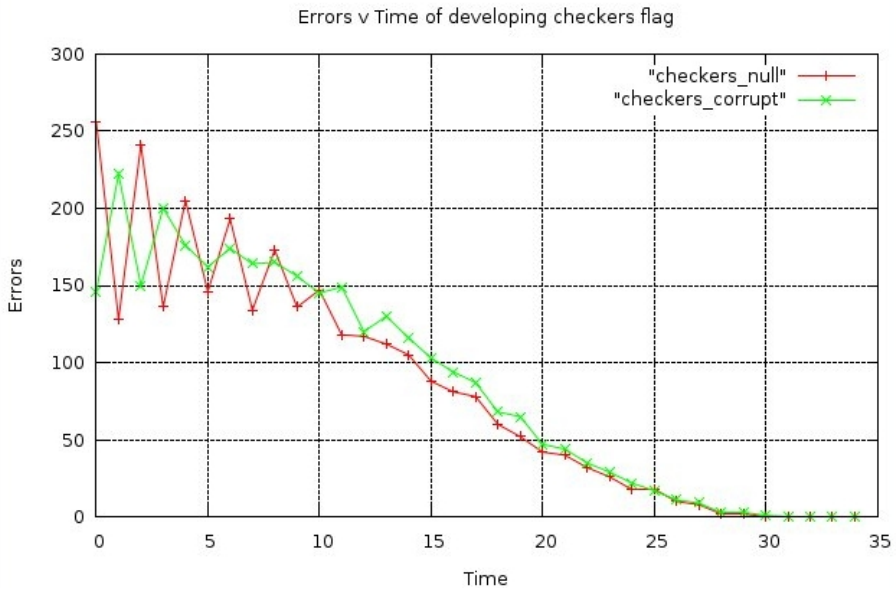


Figure 7.11: Errors v Time of developing checkered pattern

Diagram 7.12 shows the development of the 32 by 32 Welsh flag from an initial state of zero.

Diagram 7.13 shows the development of the same 32 by 32 Welsh flag from a corrupt starting state that has been formed with a random number generator.

Figure 7.14 is a graph of the number of cells with incorrect states during the assembly cycle versus time, starting from null and corrupt initial conditions.

7.5 Developing a 250 by 250 Image “Lena”

The image “Lena” is a benchmark for image processing comparisons. It has been chosen here to demonstrate the scalability of the design algorithm.

This design requires a 2236-letter alphabet and 60386 rules, almost one for each of the 62500 cells. This suggests there is little repetition within the image.

Figure 7.17 is a graph of the number of cells with incorrect states during the assembly

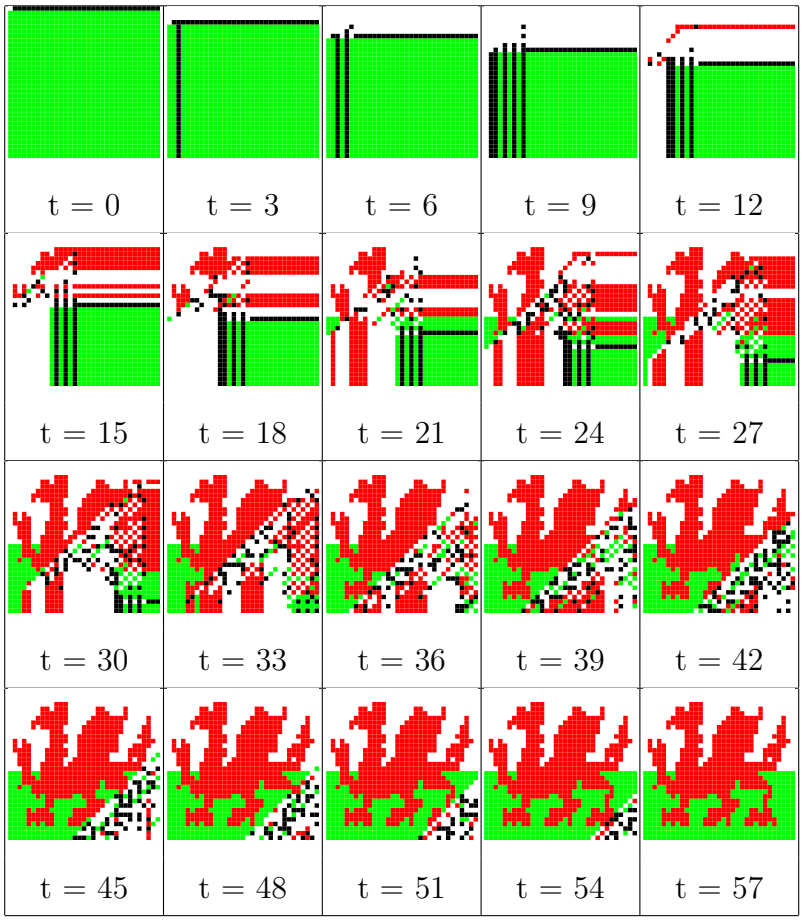


Figure 7.12: The development of a 32 by 32 Welsh flag from the null state

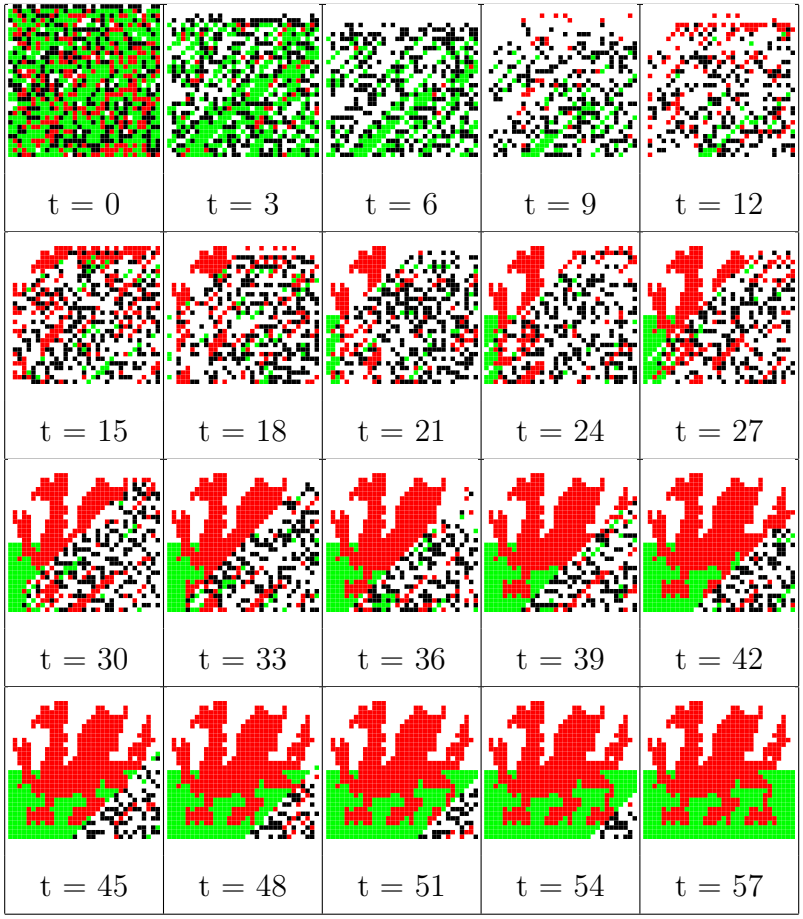


Figure 7.13: The development of a 32 by 32 Welsh flag from a corrupt state

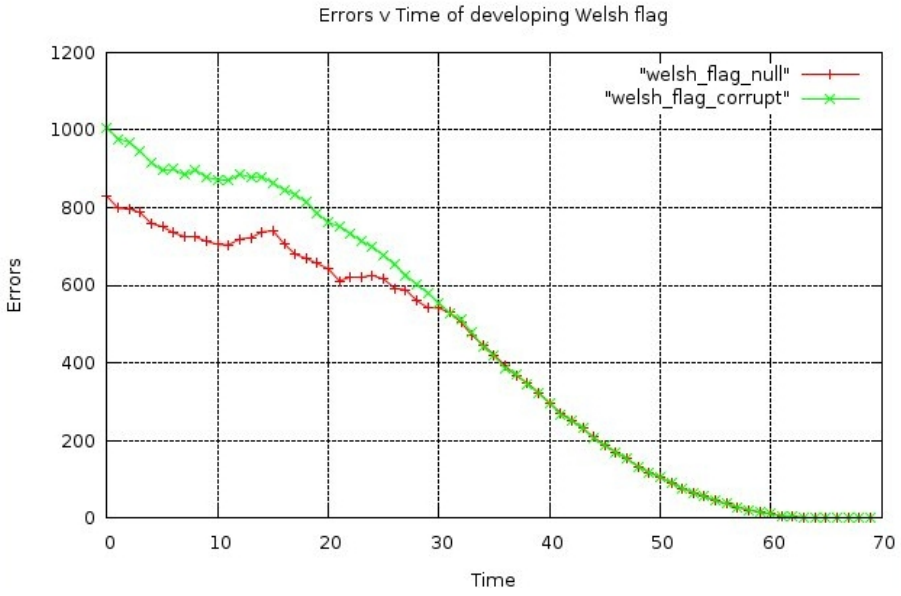


Figure 7.14: Errors v Time of developing Welsh flag

cycle versus time, starting from null and corrupt initial conditions. The corrupt error-rate closely follows the null error-rate, such that at the scale of figure 7.17 they are indistinguishable. Figure 7.18 shows the error-rate for a smaller portion of the development cycle, wherein the difference between development and repair is apparent.

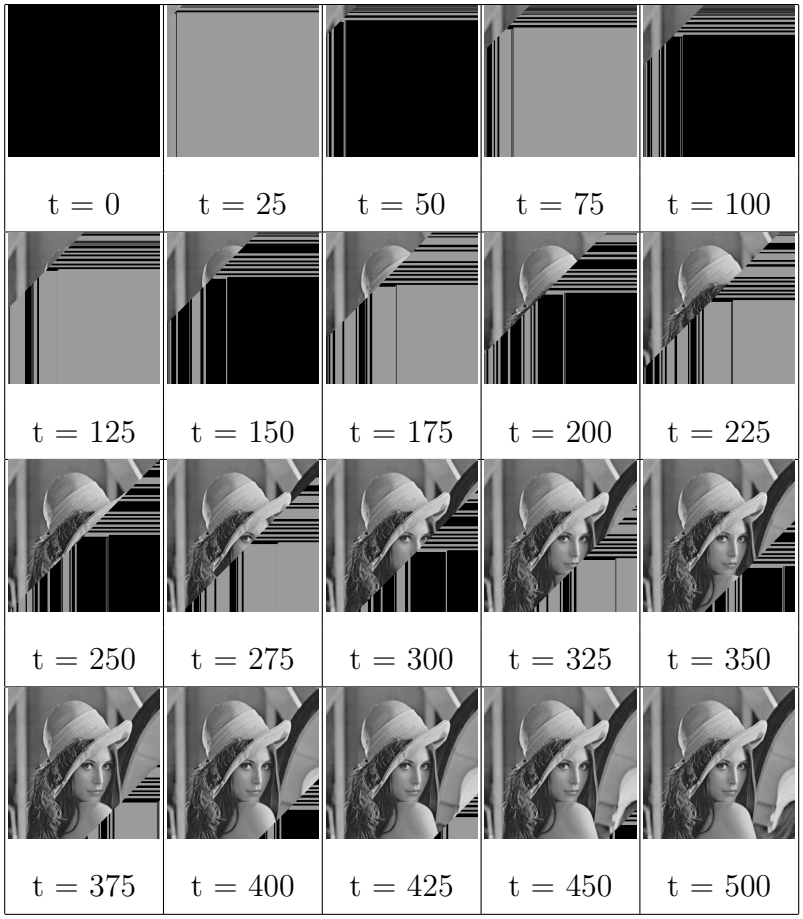


Figure 7.15: The development of a 250 by 250 greyscale image from the null state

7.6 Observations

The scheme proposed so far relies on each cell computing its next state from the current state of two neighbouring cells. In a rectangular array of cells, most will have two neighbours upon which they determine their next state, some will have one neighbour and one will have no neighbours. In effect the desired pattern emerges from this one cell (the origin cell), as shown in figure 7.19.

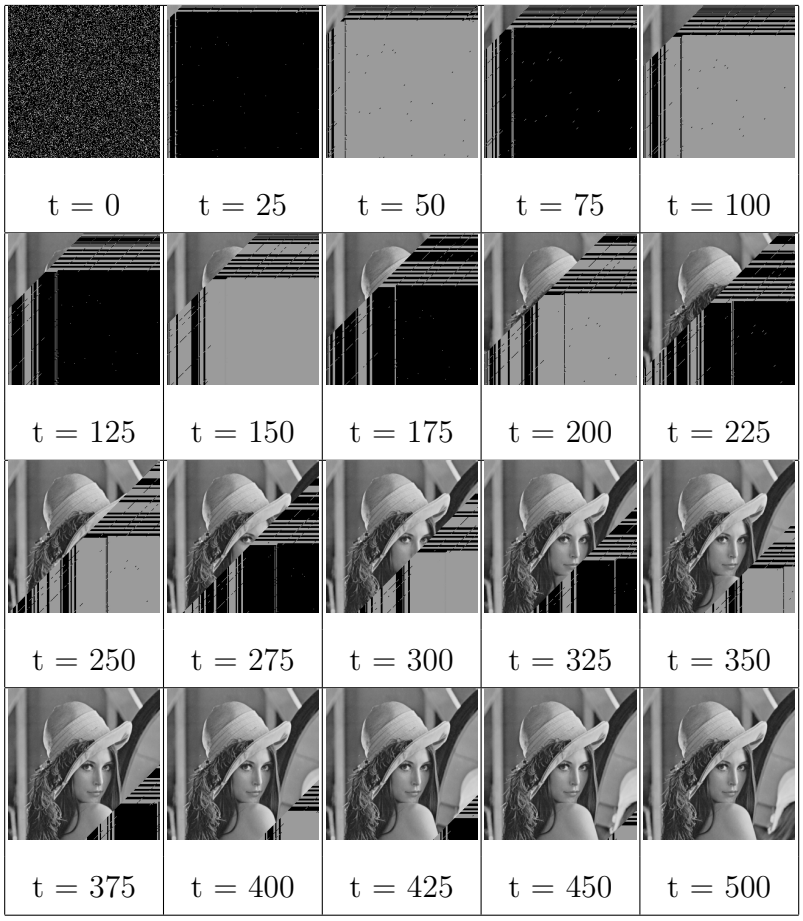


Figure 7.16: The development of a 250 by 250 greyscale image from a corrupt state

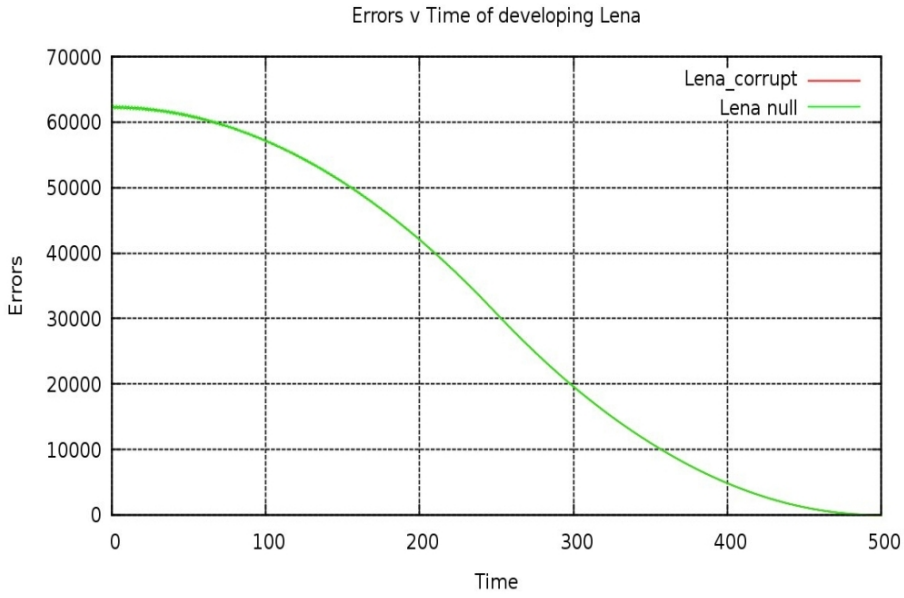


Figure 7.17: Errors v Time of developing 250 by 250 image

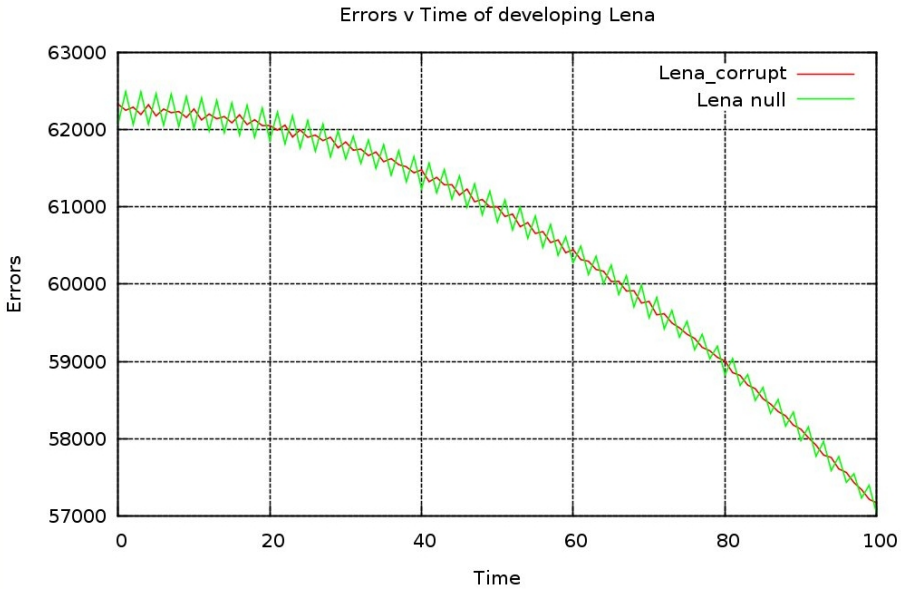


Figure 7.18: Errors v Time of developing 250 by 250 image

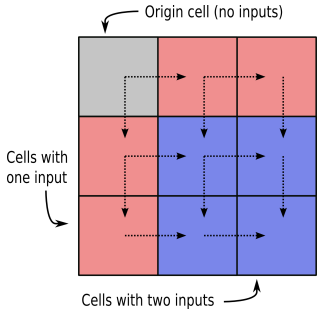


Figure 7.19: Input combinations for each cell

The pattern emerges from this origin cell in diagonal stripes orthogonal to the origin-antipode line. Thus for a four-by-four array assembling from null, the number of cells solved by each iteration versus time is the sequence 1-2-3-4-3-2-1. More generally, equation (7.3) shows the number of cells solved per iteration for a square array, equation (7.4) shows the number of cells solved per iteration for a rectangular array. s = number of cells solved, w = width, h = height, t = time.

$$s = w - |t - w| \tag{7.3}$$

$$s = \frac{w + h - |t - w| - |t - h|}{2} \tag{7.4}$$

The cumulative number of cells solved per iteration approximates to an S-curve as the size of the automata increases. This can be observed in figures 7.3, 7.7, 7.11 and 7.14.

The reason for the perturbations away from the S-curve (as best seen in figure 7.11) is the effect of the solved cells on those cells still to be solved. For these cells, the states they observe in their neighbours are transient, however certain combinations of these states will coincidentally correspond to correct cell outputs for these cells.

Two possible applications for this technique that are worth a brief consideration are:

1. **Lossless Image compression** A 100 x 60 pixel image of a checkers board can be saved as a 24K bitmap file, a 236 byte PNG file, or a 196 byte GIF file. If instead it is saved as a file that describes the rules and assignments that are needed for it to self-assemble, the result is a 40 byte file. However, this image only has two colours. The results from attempting to compress three 40K arbitrary photos can be seen in figure 7.20. Note that 'dtime' refers to the time required to design the compressed image.

Image	PNG size	Assignments	Rules	File size	Compression	dtime/s
1	54K	39125	78766	1571192	0.1	854
2	70K	39918	79916	1597680	0.1	1098
3	58K	39887	79864	1596560	0.1	875

Figure 7.20: Results from compressing 40K images

   That the file sizes are actually getting bigger (by a factor of 10) suggests this algorithm is not an effective image compression tool.

2. **Image entropy analysis** The algorithm presented in previous chapters is a means of encoding an image as the relationship between each pixel and its immediate neighbours. The number of unique relationships is inversely related to the number of large or repeated features, or repeated relationships between features. As such it is perhaps an appropriate method of evaluating the entropy of an image.

The size of a JPEG file is a function of the number of high-frequency components within an image, not their placement within the image. Thus an image of eight repeated stripes of equal size will encode to the same size JPEG as an image of eight stripes of different sizes. However, this is a simple test to estimate the entropy of an image. Figure 7.21 shows the correlation between the size of the JPEG file versus the number of rules and assignments needed to encode it. Each of the points refers to a different sample image, the lines are lines of best fit.

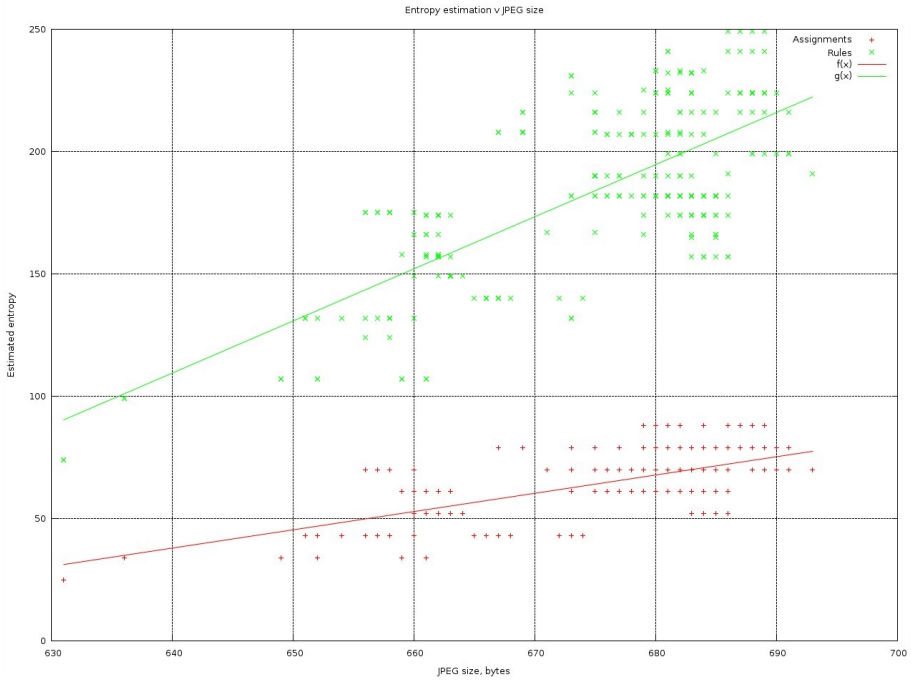


Figure 7.21: Correlation between JPEG file size and number of rules and assignments needed to encode it

Chapter 8

The community effect — a bio-inspired optimisation

Morphogenesis is the process that underpins the self-organised development and regeneration of biological systems. Professor John Gurdon determined via a series of experiments [HG90] that another mechanism assisted and complemented morphogenesis during the assembly of biological systems.

8.1 The community effect in animal development

By using a culture of *Xenopus laevis* (African clawed frog), blastula cells, Gurdon was able to induce a tissue of un-differentiated (vegetal) cells to differentiate into a concentrated block of muscle cells. Without the Xenopus stimuli, these cells would have differentiated into epidermis cells.

Attempts to repeat this differentiation with just one animal cell were unsuccessful, suggesting that an inter-community facilitator was at work within the larger community of cells, ensuring the necessary correct differentiation of its constituent cells.

Gurdon proposed an explanation, the so-called community effect. Namely that some hitherto unknown short-range inter-cellular chemical communications were responsible for re-enforcing the message of the morphogen stimuli within the community.

Figure 8.1 shows the muscle cells (in white) developed as part of this experiment, in response to the Xenopus stimuli.

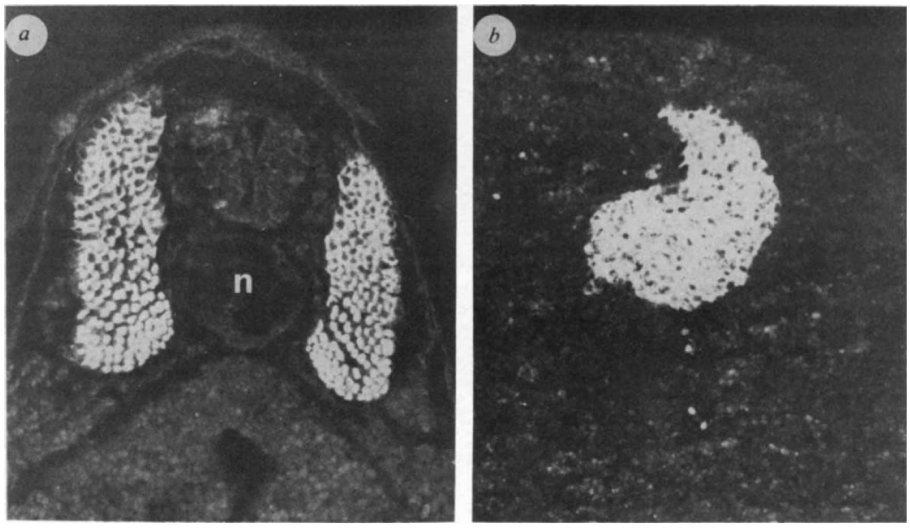
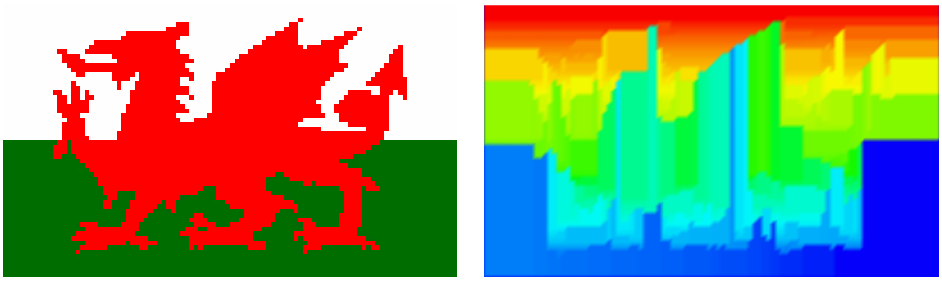


Figure 8.1: Muscle cells developed in (a) a tadpole, (b) a conjugate of animal and vegetal tissue [HG90]

8.2 A community effect model

This model is an adaption of the cellular automata models demonstrated in earlier chapters. Instead of an array of identically-sized cells, we group neighbouring cells with common output values into larger community cells. The result is an amalgamation of cells of different sized rectangles. Figure 8.2 is an example division of the Welsh flag pattern on 6000 cells divided into 560 communities.

Assigned to each community is a state. The origin cell of each community determines its next state and the dimensions of its community from an internal look-up table and the states of its neighbours. Each neighbour of this origin cell (in the direction of information propagation, e.g. north-west to south-east) then copies its



(a) (b)

Figure 8.2: (a) Welsh flag, (b) Welsh flag in 560 communities

state and community dimensions, subtracts one from the appropriate dimensions before passing this onto its neighbours. When a cell receives the dimensions of its community with a zero for either its height or width it knows it must be a new origin cell and determines its next state from its neighbours instead.

8.3 The design algorithm

The design algorithm for these heterogenous amalgamations of communities is similar to the algorithm already presented for the design of homogenous arrays of identically sized cells. It is a two-stage process, cells are grouped into communities then assigned a common state.

8.3.1 Grouping algorithm

Neighbouring cells with identical output colours must be grouped into larger rectangular communities of cells. There are two common approaches to this type of problem:

1. A greedy algorithm: For each cell, make the locally optimum choice with the hope of finding the global optimum. An implementation would be to iterate over every un-grouped cell, grouping each with as many of its neighbours as

possible. This approach is faster than alternatives, but will not necessarily produce the optimum (i.e. fewest number of communities) design.

2. An exhaustive, NP-complete, search. An implementation would be to try every possible grouping of cells, then determine which has the fewest number of communities. This approach is much more computationally intensive than a Greedy algorithm, but will produce an optimum design.

The algorithm demonstrated and evaluated here uses the faster, greedy algorithm.

8.3.2 State assignation

Each community must be assigned a state. This state maps (many-to-one) to both an output colour and the dimensions of its community. The algorithm presented in chapter five can, in principle, be used for this design. There are a few difficulties, however, with simple iteration from the south-east community towards the top-left origin community.

For instance figure 8.3(a) shows a community that starts with the cell at column one and row one, (row,col)=(1,1); the algorithm needs to assign it a state value, x, that is compatible with its dependents. This has three dependent communities, (1,2), (2,2) and (4,2) that have already been assigned states. The rules needed for each dependent community respectively will be (state above, state left)=$(x, 0) \rightarrow 2$ (where the 0 is a boundary condition), $(x, 2) \rightarrow 2$ and $(x, 2) \rightarrow 4$. Thus it can be seen there is no solution for x for the latter two rules.

	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	2	2	2	4	4	4	4
3	2	1	1	1	1	1	1
4	2	1	1	1	1	1	1

(a)

	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0
2	2	2	2	4	4	4	4
3	2	1	1	1	1	1	1
4	2	1	1	1	1	1	1

(b)

Figure 8.3: (a) Conflicting communities, (b) A solution

Figure 8.4(a) shows an example from the design of the Welsh flag. Communities (2,1), (2,4) and (4,4) form a dependent loop; none of these communities can be solved without the other two being solved first.

	1	2	3	4	5	6	7
1	0	0	0	0	0	34	34
2	20	0	0	0	0	34	34
3	20	0	0	0	0	34	34
4	3	0	0	0	118	20	118
5	2	0	0	0	118	20	118
6	4	0	0	120	128	4	4
7	0	0	5	4	164	20	20
8	0	11	122	15	164	20	20

(a)

	1	2	3	4	5	6	7
1	0	0	0	0	0	34	34
2	20	0	0	0	0	34	34
3	20	0	0	0	0	34	34
4	3	0	0	0	118	20	118
5	2	0	0	0	118	20	118
6	4	0	0	120	128	4	4
7	0	0	5	4	164	20	20
8	0	11	122	15	164	20	20

(b)

Figure 8.4: (a) Conflicting communities, (b) A solution

For both conflicts, (8.3(a) and 8.4(a)) the solution is to divide the largest partition into two smaller partitions. These solutions can be seen in figures 8.3(b) and 8.4(b).

A general strategy to overcome these conflicts is:

1. Once a community has been assigned a state it doesn't change. To change it later would require a re-assignation of state for every cell that uses it (directly or indirectly) as an input. Also this would open up the possibility of the algorithm not having an end.

2. Each community has neighbouring communities that depend on it to determine their own state. If the propagation of state information is from the north-east, these dependents are to the west and south of itself. Each community is tested: are its dependents already solved for? If yes, solve for this community's state also, otherwise try the next community.

3. Repeat to (2) for every community in the design. If no communities have been solved in this iteration, split the unsolved community with the most satisfied dependents into two, then try again.

Figure 8.5 shows the progress of this iterative group-and-solve approach designing the Welsh flag pattern. The red line is the number of communities that have been

solved. The green line is the number of communities the design has been split into.

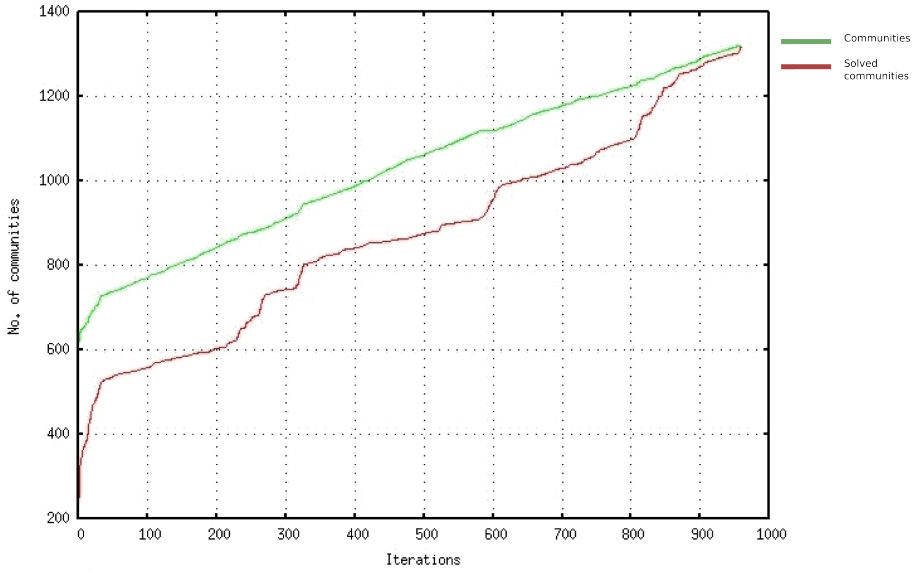


Figure 8.5: Communities created and solved per design iteration

8.4 Results

Figure 8.6 shows the communities of the Welsh-flag pattern, after the design algorithm is complete.

(a) (b)

Figure 8.6: (a) Welsh flag, (b) Welsh flag in 1103 communities

This design, and those that are presented henceforth will be compared against their equivalent created by the design algorithm presented in chapter six. The number of assignments and rules per design are a measure of how efficiently the image has been encoded. Each image is the same size (100 x 60 pixels). Rather than calculate

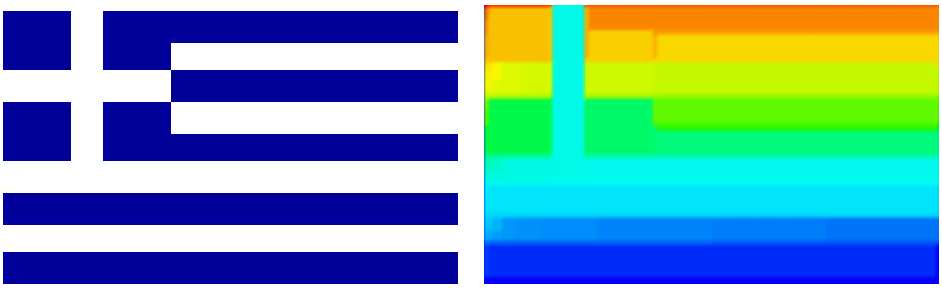
the computational complexity of each algorithm, the time each design takes to be completed is measured for both algorithms as implemented on a 3.2GHz Pentium Xeon processor.

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
3747	7829	50	1103(5, 510)	299	263	536

The number of assignments and rules needed for this improved algorithm is greater than a factor of ten smaller than those of the algorithm in chapter five. However the design time required is greater than a factor of ten greater.

Further examples (the flags of Greece, Czech Republic, Canada, the United Kingdom and the United States of America) have been chosen for their increasingly complex designs.

Figure 8.7 shows the Greek flag divided into 128 communities.



(a) (b)

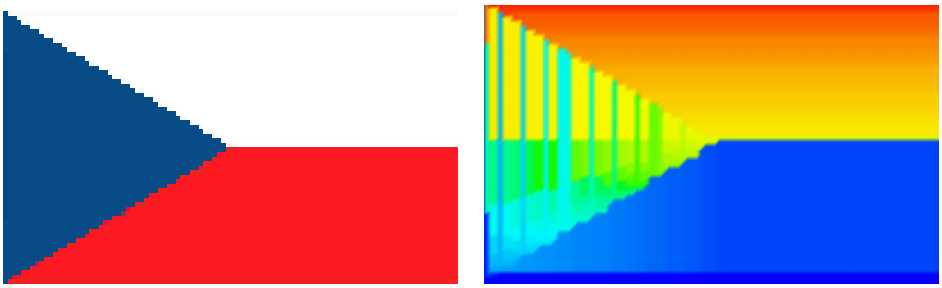
Figure 8.7: (a) Greek flag, (b) Greek flag in 128 communities

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
429	1104	4	128(47, 686)	107	86	25

For the Greek flag, the improved encoding efficiency over the original algorithm is lower. This is because the flag is formed from simple rectangles of alternating colour.

Figure 8.8 shows the flag of Czech Republic divided into 285 communities. Whilst a simpler design than the flag of Greece, the presence of diagonals prevents the design

from being split into a small number of rectangles.



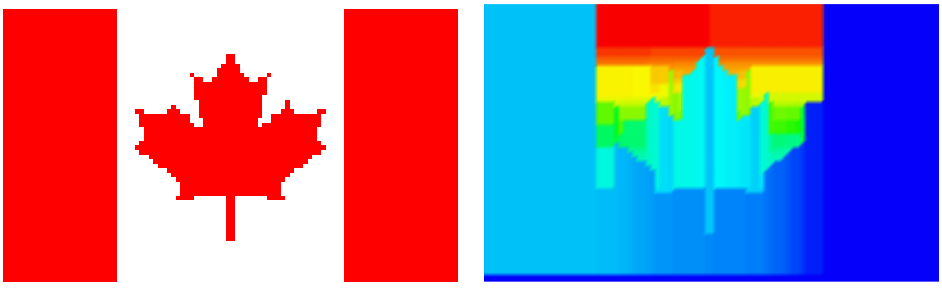
(a) (b)

Figure 8.8: (a) Czech Republic flag, (b) Czech Republic flag in 285 communities

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
2408	5032	27	285(21, 1322)	181	124	26

The number of assignments and rules needed for this algorithm is greater than a factor of ten improvement over the old algorithm. The design time is also comparable.

Figure 8.9 shows the flag of Canada split into 143 communities. The flanking red bars that form two-thirds of this pattern need just two communities to encode, however the maple leaf in the center is more complicated, requiring 141 communities.



(a) (b)

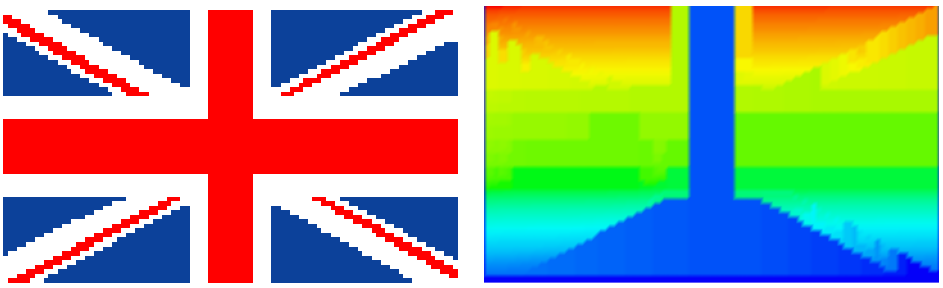
Figure 8.9: (a) Canadian flag, (b) Canadian flag in 143 communities

Again a ten-fold improvement in coding efficiency is demonstrated. Of note is that the design time for the improved algorithm is also shorter than that of the original. This is because two-thirds of the design (4000 cells requiring 4000 state assignments

and rule calculations in the original algorithm) can now be solved with just two state assignments and the creation of two rules.

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
839	2571	15	143(42, 1500)	76	58	9

Figure 8.10 shows the flag of the United Kingdom divided into 772 communities. The presence of diagonals in this design prevents the design from being split into a small number of rectangles.



(a) (b)

Figure 8.10: (a) United Kingdom flag, (b) United Kingdom flag in 772 communities

Again a ten-fold improvement in the coding efficiency is demonstrated. The design time is 1.5 times that of the original.

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
3832	8482	117	772(8, 600)	219	187	187

Figure 8.11 shows the designed automata converging to the pattern of the UK flag in 150 iterations. Figure 8.12 shows the same automata converging from a random starting state to the pattern of the UK flag.

Figure 8.13 shows the flag of the United States divided into 480 communities. The stripes form large rectangles while the stars form many, much smaller, communities.

Here the efficiency of coding improvement is not quite ten-fold, but the design time is more than 75 times greater.

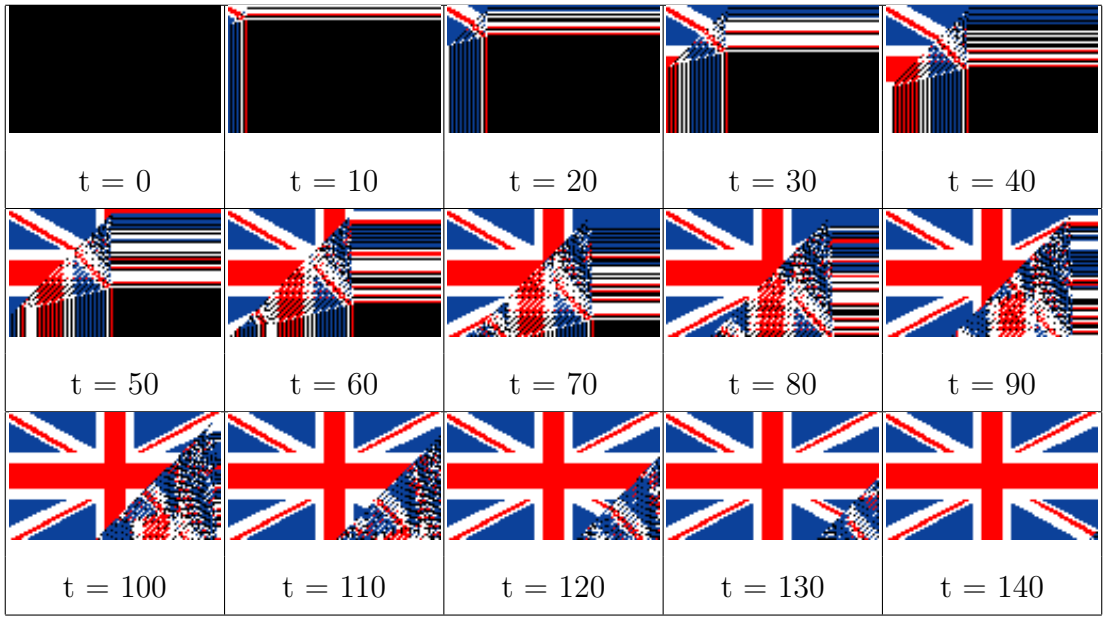


Figure 8.11: UK flag assembling from null over 140 iterations

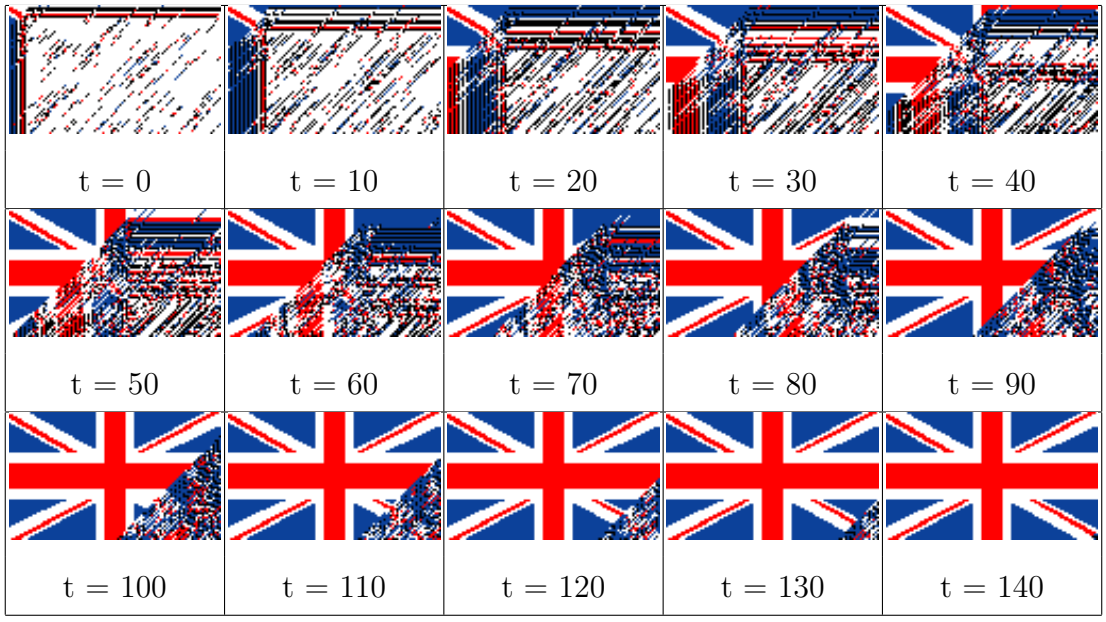
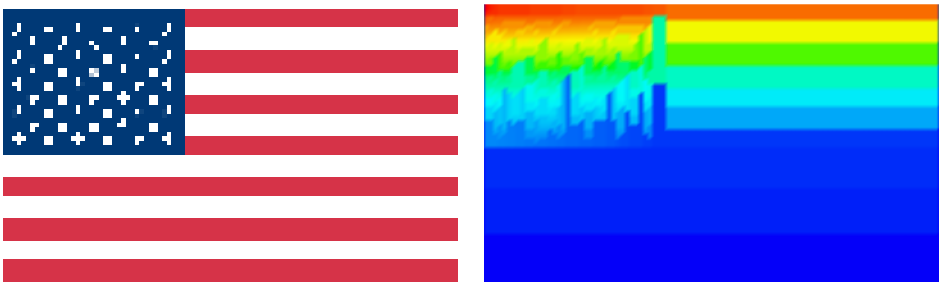


Figure 8.12: UK flag assembling from random over 140 iterations



(a) (b)

Figure 8.13: (a) United States flag, (b) United States flag in 480 communities

Original design algorithm			With n communities of x_i cells			
Assignments	Rules	time/s	$n, (\overline{x}, max(x))$	Assignments	Rules	time/s
558	1616	7	480(13, 500)	82	74	527

Figure 8.14 shows the designed automata converging to the pattern of the US flag in 150 iterations. Figure 8.15 shows the same automata converging from a random starting state to the pattern of the US flag.

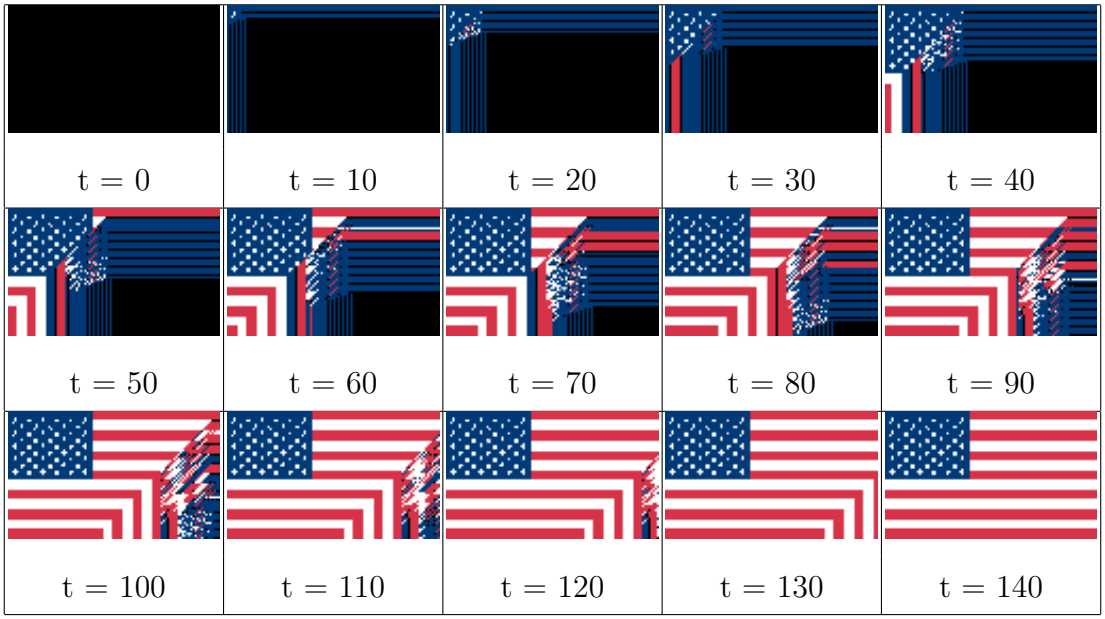


Figure 8.14: US flag assembling from null over 140 iterations

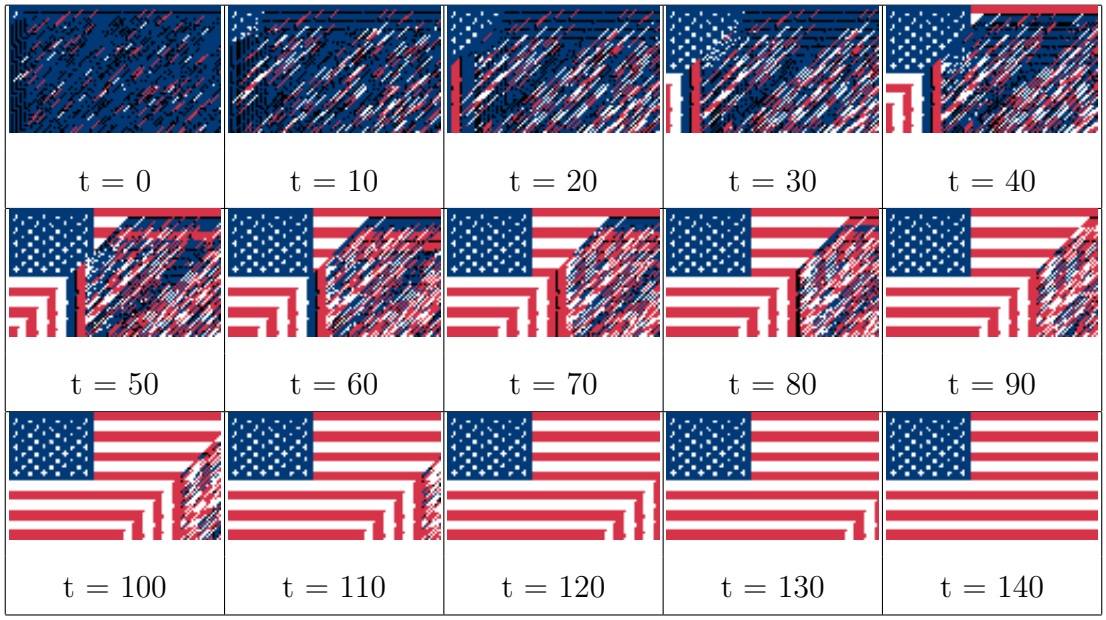


Figure 8.15: US flag assembling from random over 140 iterations

8.5 Observations

Presented in this chapter is a further refinement of the design algorithms presented in the preceding chapters. This refinement is a minimalist model of a technique used by biological systems to locally re-enforce the messages diffused by morphogenesis.

This improvement requires a more complicated design algorithm and a more complicated assembly algorithm, but the design rules and assignments more efficiently encode the intended global pattern.

.

Chapter 9

Design for reliability: analysis and techniques

> Reliability is, after all, engineering in its most practical form. *James R. Schlesnger (Former US Secretary of State for Defence)*

The consumer will not tolerate electronic products that are prone to failure: cars such as the Skoda Felicia or the Ford Pinto, motorbikes such as the BSA Dandy scooter, computers such as the Apple III - these devices were all commercially unsuccessful simply because of their high failure rate. For a product to succeed, the designer must consider the intended lifespan of the device and then maximise the probability of its survival for that period.

Within the systems upon which we depend, or choose to trust, for our survival - be they aeroplanes, trains, armaments or systems critical to a country's infrastructure - failure cannot be tolerated without adequate notice and provision for repair or replacement. 500 people died when Pfizer's replacement heart valves failed, costing the company upwards of $200 million. The structural failures of the de Havilland Comet jet aeroplane caused five crashes before commercial flights were cancelled. A designer must therefore consider the significance of a component failure and make provisions for repair or replacement.

Components that are too costly to repair or replace, because of limited accessibility or limited resources, cannot be subject to failure. The early operational failures of the Hubble telescope were due to a flawed mirror. A space mission costing $8 million was required to fix the telescope with correcting lenses. Thus for a system to succeed, a designer must consider the cost of repair when deciding on the required level of reliability.

The British Post Office started the design for reliability trend. As owners of a Mark I telephone exchange that used thermionic valves, the Post Office was plagued by failure. In an effort to reduce the number of failures, and thus the running costs, the department in charge of the exchange started collecting statistics on the number and type of failures per week and discovered that, by leaving the valves constantly turned on, the number of failures decreased.

Mobile computing platforms are the pervading technology of the 21st century. Every device is a compromise of size, battery life, functionality and reliability. Connect [Con05] reviewed the reliability of 38 different mobile phone models available in 2005. Figure 9.1 shows the proportion (grouped by manufacturer) of each that had to be repaired after completion of the test cycle.

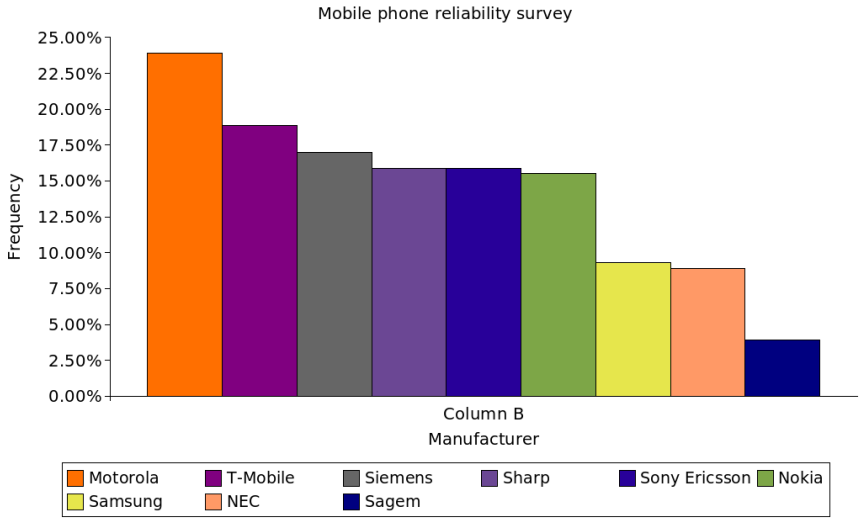


Figure 9.1: A comparison of mobile phone reliability

Image courtesy of Connect Magazine

The current trend is for new devices to be smaller and more functional than their predecessors. Modular redundancy, the use of extra copies of failure-prone hardware that can mask, or take the place of its damaged counterparts, has costs in both size and functionality. If this trend continues, the price of modular redundancy will become greater and the reliability of mobile devices will decrease.

Aside from demanding applications, reliability engineers face a challenge from systems formed on unreliable mediums. Plastic electronics systems are formed on flexible substrates, typically by the sequential deposition of conductive or semiconductive organic polymers (such as Poly-3,4-ethylenedioxythiophene) using analogue or digital graphic printing technologies. Potential applications include flexible displays, photovoltaics that can cover non-planar surfaces, and smart packaging, including battery testers, flexible batteries and RFID tags. However, flexible substrates are prone to distortion, potentially compromising the deposited system and causing its failure. Redundancy is particularly well equipped to cope with the failure of known unreliable sub-systems, but is much less well suited to coping with a system of which any part, or combination of parts, is likely to fail.

9.1 Ultra reliability

Redundancy, in its various forms, provides the reliability engineer with a versatile set of tools for designing systems to be reliable. Despite this, there remain applications to which such tools might be inadequate. Since 2002 NASA have been running a series of workshops under the title "Ultra reliability"; this with the goal of increasing systems reliability by an order of magnitude across complex systems, hardware (including aircraft, aerospace craft and launch vehicles), software, human interactions, long life missions, infrastructure development, and cross cutting technologies [Sha06]. It is not difficult to see the difficulties standard redundancy techniques will face in long life missions (a manned mission to Mars will take upwards of six months, the $720 million Mars Reconnaissance Orbiter has a planned life of at least

four years) that are vulnerable to cosmic rays.

The following case example describes the application of "Ultra reliability" to the design and assessment of a safety-critical system of the Large Hadron Collider (LHC). The reliability of this system was assessed by the author whilst based at CERN, and is quoted here with their permission.

The LHC Beam Dump System (LBDS) has been designed to achieve a set of demanding safety and reliability standards. The role of the LBDS is to safely extract both beams from the LHC into two graphite blocks. The LBDS consists of 15 horizontally deflecting extraction kicker magnets (MKD), one superconducting quadrupole Q4, 15 vertically deflecting septum magnets (MSD), ten dilution kicker magnets (MKB) and several hundred meters of transfer line up to a dump absorber block for each ring (LHC). The LBDS is located next to the CMS detector of figure 9.2. The LBDS system can be seen in figure 9.3

The Beam Energy Tracking System (BETS) is responsible for monitoring the energy of the hadron beams within the collider tubes and setting the MKD power levels accordingly. A failure of the BETS system could result in a 7TeV beam colliding with precision, expensive and difficult to replace equipment that immediately surrounds the collider tubes. This is why the specified failure rate is less than 1 failure per 10^9 operating hours. To achieve this reliability, a static redundant system equivalent to 108-modular redundancy is used; as such the predicted hazard rate is $2.89.10^{-313}$ failures per year which is within the specified acceptable limits.

This chapter will present a series of tools for the design of electronic systems that self-assemble and self-repair. These tools are inspired by the study of morphogenesis detailed in previous chapters.

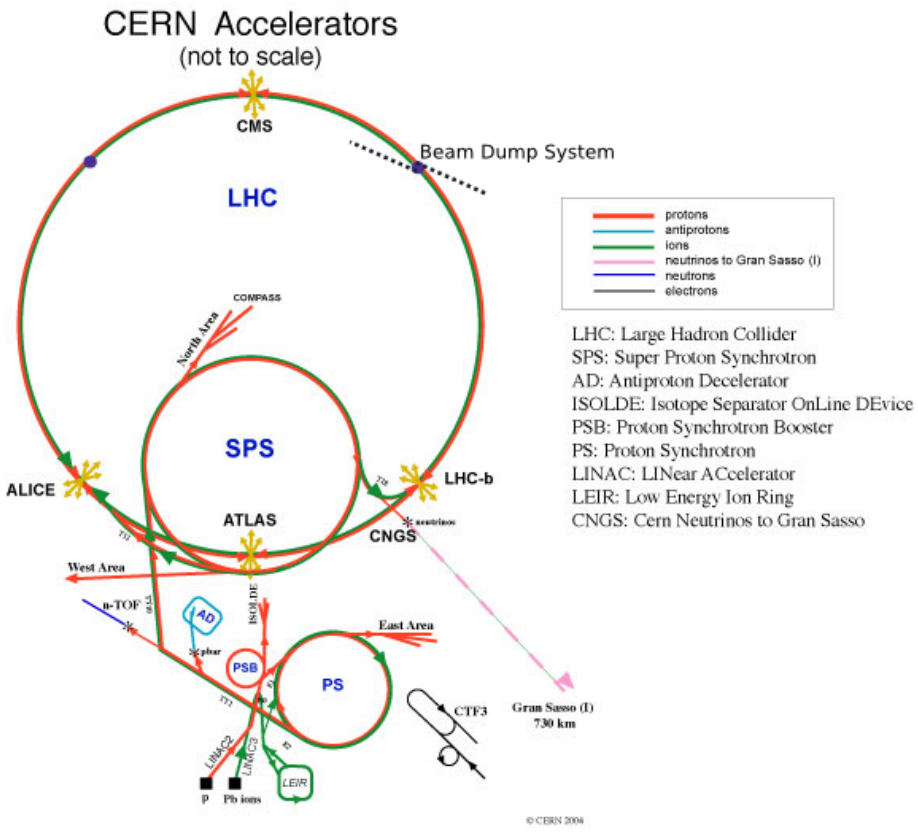


Figure 9.2: The Large Hadron Collider

Image courtesy of CERN

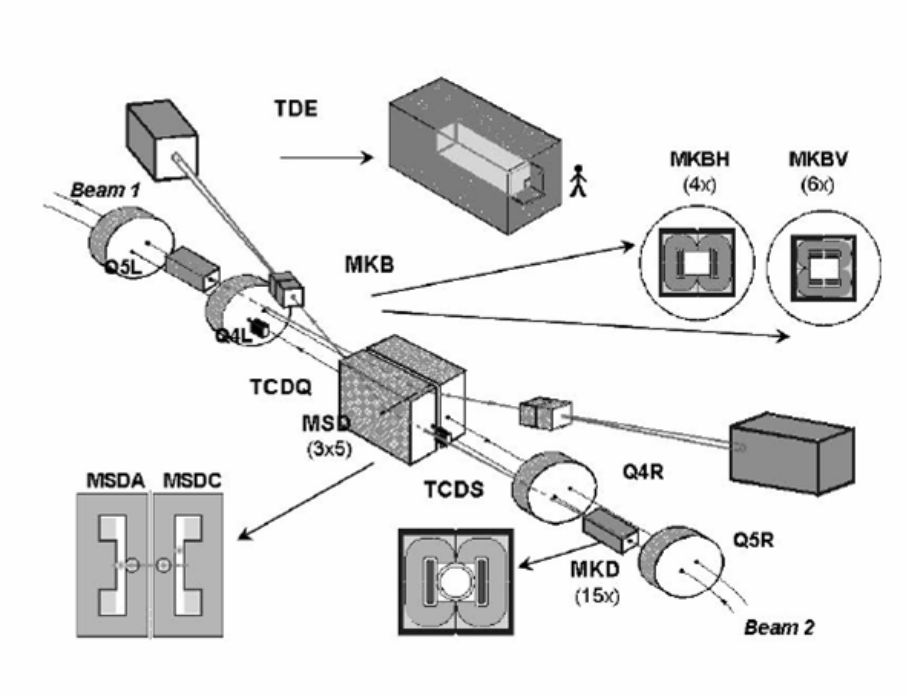


Figure 9.3: The LHC Beam Dump System

Image courtesy of CERN

9.2 Reliability analysis

In order to compare the reliability of any morphogenesis-inspired tools with that of existing techniques, we need both metrics to describe failure and tools for analysing the probability of failure [Fil05].

The following terms are used to describe the failure of a system:

1. **Failure**: the deviation of a system, or component of, from the specified behaviour. Failures are called "random" if they occur due to progressive degradation (hardware), and "systematic" (hardware and software) if they are introduced during the system life cycle [IBN96]. A progression of component failures and their consequent propagations is the chain of events that typically causes system failure. This is best described by the fault-error-failure model, known as the "chain of threats" [BR90]

2. **Error**: refers to a system state that is contrary to the design. An erroneous state may result in a failure.

3. **Fault**: that which causes an error. In software this is a mistake in the code. In hardware this could be a broken track. Reproducible faults are called permanent faults, and move the component into a persistent faulty state. Faults are called transient if they happen under conditions that are difficult to reproduce and predict.[Fil05]

Of the available metrics that relate the frequency of failure to the performance of a system, only some are applicable to any given system. This subset is determined by the consequences of a failure and by the possibility of repair - be it self-repair or repair by an external influence. For systems where one failure is one failure too many, the following metrics are applicable:

1. **Reliability, R(t)**: the probability of correct service continuing till time, t. This can be determined from the system failure rate, λ, and equation (9.1) in the case of the commonly used exponential failure distribution.

$$R(t) = e^{-\lambda t} \tag{9.1}$$

2. **Hazard rate, h(t)**: the instantaneous probability of the first and only system failure occurring.

$$h(t) = \frac{\lambda(t)}{R(t)} \tag{9.2}$$

3. **Mean time to failure (MTTF)**: the approximate amount of time expected before a system or component fails; see "mean time between failures". This is also referred to as the "life expectancy" of a system.

$$MTTF = \frac{1}{\lambda} \tag{9.3}$$

For systems that can be repaired after suffering a failure, the aforementioned metrics are less appropriate than:

1. **Mean time to repair (MTTR)**: the total corrective maintenance time divided by the total number of corrective maintenance actions during a given period of time.

2. **Mean time between failures (MTBF)**: An indicator of expected system reliability calculated on a statistical basis from the known failure rates of various components of the system.

3. **Availability**: the degree to which a system is operable and is in a committable state at the start of a mission, when the mission is called for at an unknown (i.e. a random) time.

$$Availability = \frac{MTBF}{MTBF + MTTR} \tag{9.4}$$

4. **Maintainability**: a characteristic of design and installation, expressed as the probability that an item will be retained in, or restored to, a specified condition within a given period of time, when the maintenance is performed in accordance with prescribed procedures and resources.

5. Failure rate, λ: the frequency with which an engineered system or component fails, expressed, for example, in failures per hour. If the MTBF is constant,

$$\lambda = MTBF^{-1} \tag{9.5}$$

9.2.1 Modelling component failure

The Weibull distribution

The Weibull distribution is a continuous probability distribution with a probability density function of [Wil]:

$$f(t) = \frac{\beta}{\eta}(\frac{t-\gamma}{\eta})^{\beta-1} e^{-(\frac{t-\gamma}{\eta})^{\beta}} \tag{9.6}$$

Where:

1. β is the shape parameter
2. η is the scale parameter
3. γ is the location parameter

Chiefly due to its flexibility, the Weibull distribution is often used to model the hazard rate of systems throughout their respective lifetimes. The distribution can imitate many other distributions, including the normal and exponential, making the Weibull distribution a powerful tool for reliability analysis.

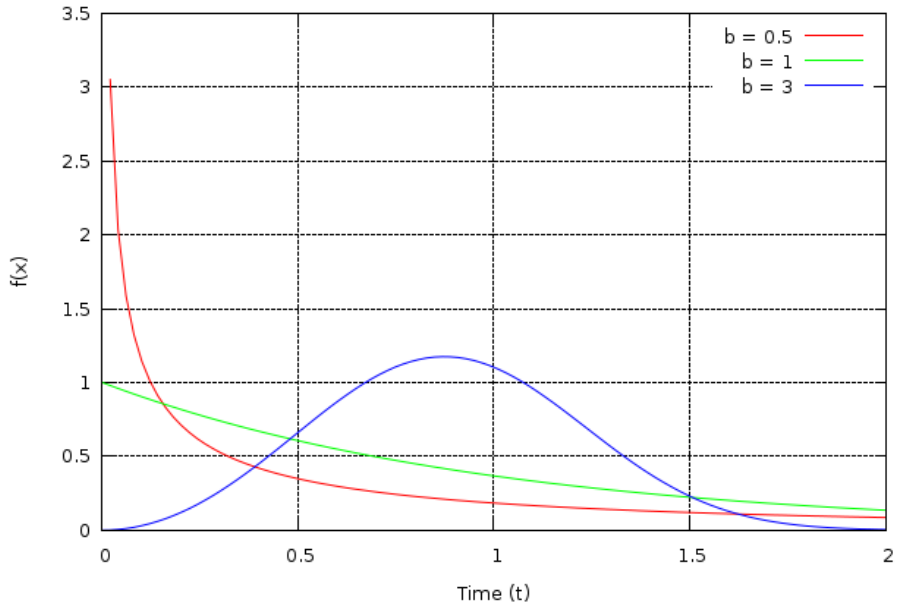


Figure 9.4: The effect of the shape parameter on the Weibull distribution ($\eta = 1$,$\gamma = 0$)

The bathtub curve

The lifetime of a product can be divided into three stages, “burn-in”, “useful period” and “wear-out”. Each stage has its own associated failure mechanism. Together they form the “bathtub curve” - a description of the lifetime of a population of products of the same design.

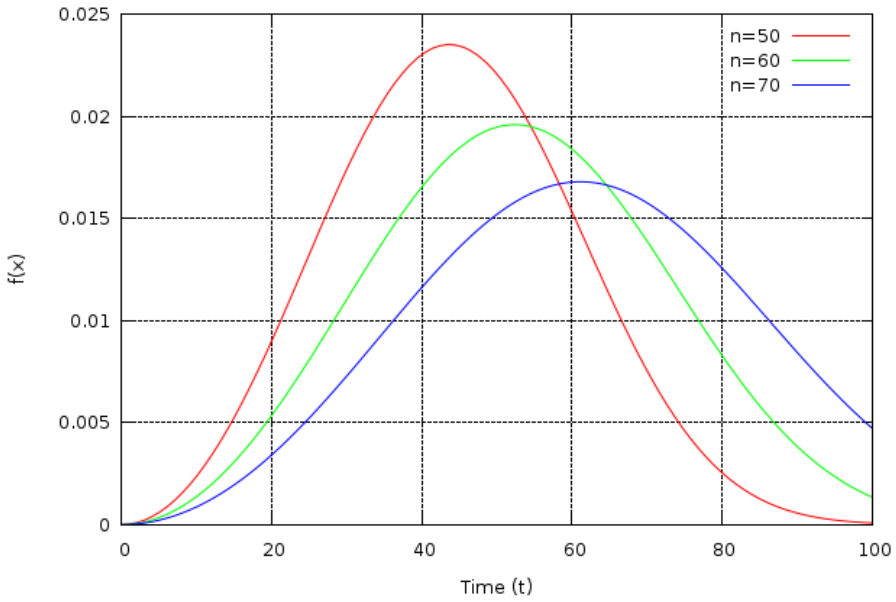


Figure 9.5: The effect of the scale parameter on the Weibull distribution ($\beta = 3$,$\gamma = 0$

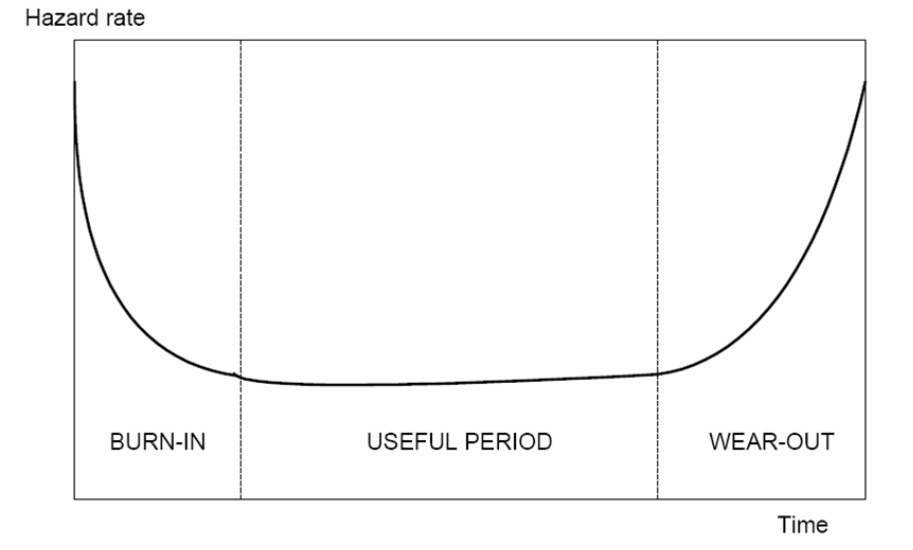


Figure 9.6: A characteristic model of the product lifetime, the bathtub curve

Infant mortality

"Dead on arrival" products are a consequence of the high failure rate that dominates the first, and normally shortest, period of the lifetime of a product. This failure rate is the result of manufacturing defects, be they errors in the assembly, defects in the material or the constraints of the assembly line. Integrated circuits are one example of a product population with a very high infant mortality rate; see table 9.7 for typical values.

Product	Area(mm squared)	Yield
386DX	43	71%
486DX2	81	54%
PowerPC 601	121	28%
HP PA 7100	196	27%
DEC Alpha	234	19%
SuperSPARC	256	13%
Pentium 3	296	9%

Figure 9.7: Yield of semiconductors [Gwe93]

As the product ages, the rate of hazards due to manufacturing defects decreases. A characteristic model of this period of the life of a product is a Weibull distribution with a beta less than 1. Typical values of a beta are between 0.2 and 0.6.

"Dead on arrival" products, from the point of view of a customer, are unacceptable. Thus to combat this phenomenon a "burn-in and test" phase is often incorporated into the manufacturing process. Burn-in is the process of exercising a product prior to its release, the intent being to age the component beyond its infant-mortality period. To facilitate quick accomplishment this process typically involves operating conditions that are designed to accelerate the ageing process. This might involve high or rapidly alternating temperatures, a vibration test and an exhaustive test of all system states.

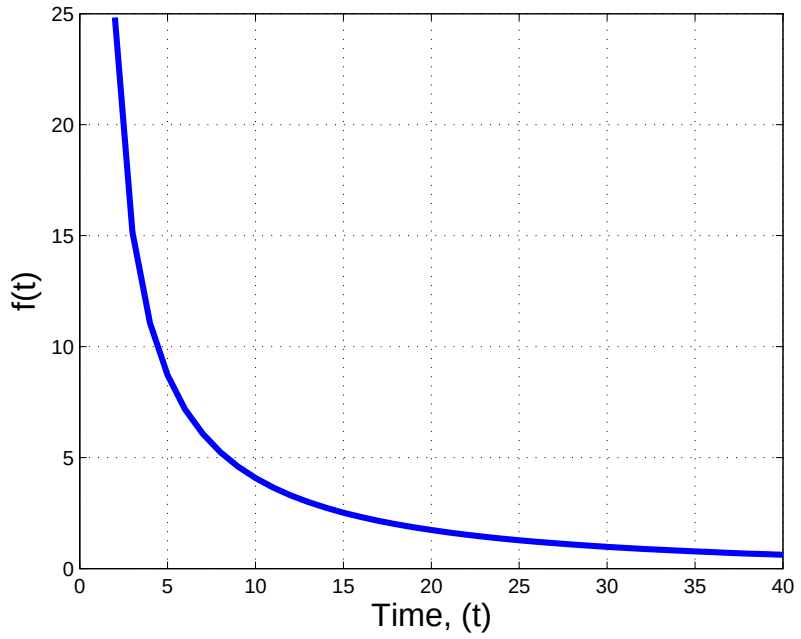


Figure 9.8: A characteristic model of infant mortality, a Weibull plot with beta = 0.2

Intrinsic failure

Subject only to truly random failures, the second phase of the lifetime of a product is characterised by its constant failure rate. Many such failure modes exist, but a good example is the soft error rate (SER) of solid state memory chips.

SER is the rate of corruption of data stored in solid state memory. If a bit is flipped from its previous value due to external, unintended conditions, the dataset of which it is a part is corrupt. If that bit is read prior to it being corrected, the error moves from being latent to being active, and a potential failure mode is created. The flipping of a bit can be caused by electro-magnetic interference, cosmic rays or alpha particle collisions, and is characterised against time as a random distribution, the sum of which is a constant failure rate.

Little can be done to prevent SER. A component can be partly shielded from electro-magnetic radiation, cosmic rays and alpha particles, but at significant cost and with limited effect.

Wear-out failure

Electromigration, corrosion, thermal separation of layers or contacts; these are but a few mechanisms by which an electronic component can wear-out. For mechanical components wear-out is the primary failure mode. Often wear-out is expected and not normally considered a failure, the cartridges of inkjet printers eventually run out of ink and need replacing, for instance. The shortest-lived critical component will determine the lifetime of the product. The MTBF parameter typically refers to the wear-out probability of failure. The cumulative MTBF plots can be characterised by a Weibull distribution with a beta that is greater than 1.

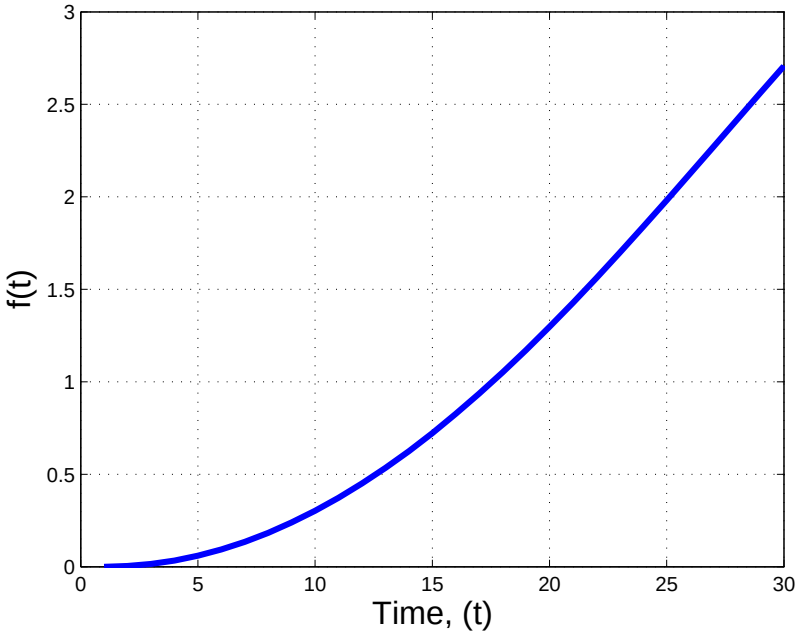


Figure 9.9: A characteristic model of wear-out: a Weibull plot with beta = 1.4

The product lifetime

The product lifetime, as described by the bathtub curve, is produced by the superposition of the Weibull distributions that characterise the first and third stages of the lifetime of the product on to the constant hazard rate of the second stage.

9.2.2 Assessing the reliability of systems

The TTF distribution of a component is estimated from failure reporting, during the system use, or from reliability runs and accelerated life testing [AR80] [Sho68]. Alternatively, the TTF of a component or system can be calculated from known component failure rates. The most popular reliability prediction tool is the Military Handbook 217F. Various versions have been published by the Department of Defence of the United States [DOD90], and more recently by the Reliability Analysis Center (RAC) [RAC97].

MIL-HDBK-217F standard

This standard consists of models for various electronic components to predict failure rates according to a number of parameters that include environmental conditions, quality levels and stress conditions.

There are two methods used as part of this programme: a part stress analysis and a parts count.

Part stress analysis

The failure rate is calculated as a function of various parameters: environmental and operational temperatures, humidity, electrical fields, vibrations, radiations, voltage and current ratings and power and quality [PN94]. The models used vary for different part types, but are derived by analogy with a chemical reaction described by the Arrhenius equation (9.7) [Sho68].

$$TTF = Ce^{E_A kT} \tag{9.7}$$

Where:

1. C is the pre-exponential factor
2. E_A is the activation temperature of the component failure mode
3. k is Boltzmann's constant
4. T is the temperature

Equation 9.8 is the model for a field-programmable gate array (FPGA).

$$\lambda_p = (C_1\pi_T + C_2\pi_E)\pi_Q\pi_L \tag{9.8}$$

Where:

1. λ_p = FPGA failure rate
2. C_1 = Die complexity failure rate
3. C_2 = Package failure rate
4. π_T = Temperature factor
5. π_E = Environment factor
6. π_Q = Quality factor, related to the level of screening of the device
7. π_L = Learning factor, related to the maturity of the device

Parts count

This method requires less information than the part stress analysis and is thus most applicable early in the design phase. A base failure rate λ_b is calculated from the stress analysis equations for standard operating conditions, then it is adjusted with

respect to the quality factor and the environment [Bow92]. According to this method the system failure rate is given by:

$$\lambda = \sum_{i=1}^{n} N_i \lambda_i \pi_{Qi} \tag{9.9}$$

Where:

1. n = Number of part categories
2. N_i = Quantity of the ith part type
3. λ_i = Failure rate of the ith part type
4. π_{Qi} = Quality factor of the ith part type.

Many criticisms exist about reliability prediction using the MIL-HDBK 217F:

1. The assumption of a constant failure rate bounds the applicability of the tool to the useful period, which for many components ranges between two and five years only [Eco04].
2. The failure rate is not apportioned in failure modes [Bow92].
3. The failure rates database quickly becomes obsolete [Fil05].
4. The stress analysis, based on the Arrhenius equation, is somewhat arbitrary [Wat92].

Despite this, most complex systems have been certified using the MIL-HDBK-217F, including the international space station (ISS), civil and military airplanes, avionics systems and nuclear and chemical plants [Fil05].

Failure mode, effects and criticality analysis (FMECA)

A failure mode, effects and criticality analysis considers every constituent component of a system for its probability of failure, failure mode and the subsequent consequences of this failure. A resistor can fail open circuit, short circuit or drift in resistance. According to the RAC FMD-97[RAC97] handbook, of all resistor failures, 80% will be to an open circuit, 10% to a short circuit and 10% to a resistance drift. This data, combined with that of the MIL-SPEC-217F [DOD90] handbook, states that the hazard rate of a resistor failing to any one of the above failure modes is one per million hours.

The criticality of a specific component failure is a product of the probability that the component failure will cause a loss of system function and the amount of time the component is operating.

A typical reliability analysis will use FMECA in conjunction with either a Markov model or a fault tree: the FMECA to determine the probability of each Markov state or fault tree leaf event occurring, and the system model to describe the consequences of each ancillary failure event occurring.

9.2.3 Modelling failure modes

Reliability models are a function of system architecture and component failure statistics derived from techniques such as the parts count. Such models are then analysed against a description of the operating environment of the system, including the period of time for which correct performance is expected. There are two categories of models, combinatorial and state based.

Combinatorial techniques

Combinatorial techniques describe the system failure as the logic combination of failures occurring in its components. The basis is the structure function, $\phi_A : X \rightarrow [0, 1]$, a function of the state of the components X of the system arranged with respect to the system architecture A, which returns the value zero if failed, or one if functioning [Sho68].

The definition of the structure function may be applied at a lower level provided that each component is given a binary variable x that refers to its state, failed or functioning. As an example, a non-fault tolerant architecture will be sensitive to the failure of every component, as modelled with a series combinatorial model. Conversely, a redundant architecture will be sensitive to the accumulation of failures, as modelled by a parallel combinatorial model. [Fil05]

The structure function can be disseminated into minimal sets of paths, p, and cuts, χ.

A minimal path set is the set of components that are all necessary for the system to function. Thus the structure function is described by (9.10).

$$\phi_A(X) = 1 - \prod_k (1 - p_k) \tag{9.10}$$

A minimal cut set is the set of components whose failure leads to the failure of the system. Thus the structure function is described by (9.11)

$$\phi_A(X) = \prod_k (\chi_k) \tag{9.11}$$

Fault tree analysis

A fault tree is a Boolean expression of which the constituent elements assume one of two values 0,1. The root of the fault tree is the system failure. A fault-tree represents the system as a sum of minimal cut sets.

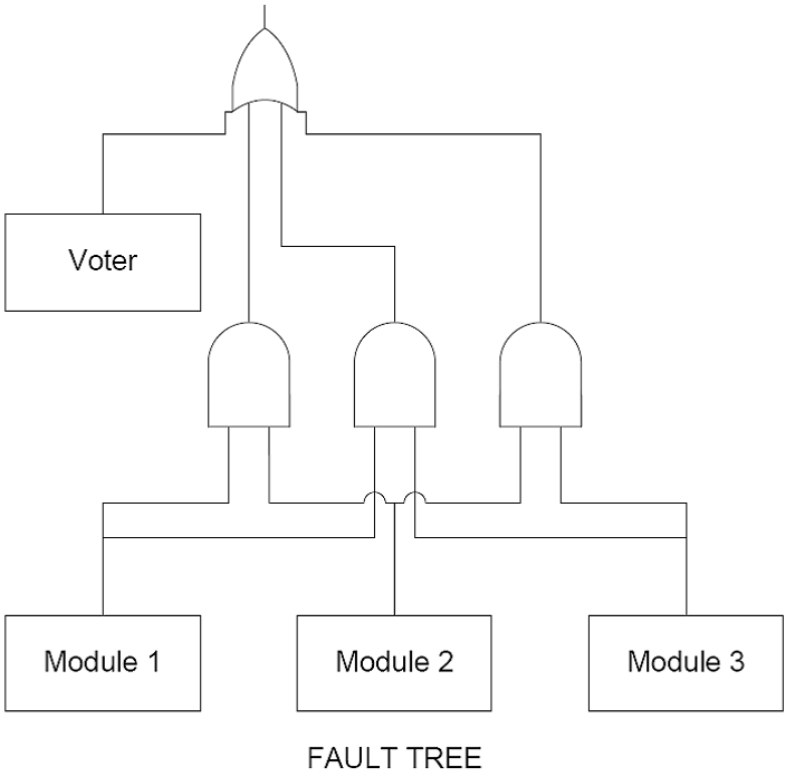


Figure 9.10: An example fault tree [Fil05]

State based models

A Discrete Events System (DES) is a stochastic timed automaton, which is formally defined with a five-tuple set $\{X, E, \delta(x, e), x_0, \psi\}$ where X is the space of states, E is the space of events, $\delta(x, e)$ is the state function that calculates the state transition given the event e has occurred, x_0 is the initial state and ψ describes the distribution functions for the events in E. [CL99].

Markov modelling

A Markov chain is a stochastic model that describes a DES. It can be used to predict the probability distribution of a system governed by a stochastic process provided the Markov assumption holds, namely: the future evolution of the state is independent of the past and only depends on the present. Note, with respect to the fault-tree analysis, the Markov model provides the probability distribution at time t, in the space of states of the system, while the fault-tree gives the probability for just one state.

For a finite state space X of size N, the total probability to be in state x_k at time t, given the initial state x_1 at time t_1, is described by the Chapman-Kolmogorov equation [Tri82].

$$P[\mathbf{X}(t) = x_k] = \sum\nolimits_{i \in X} p_{ik}(t, t_0) P[\mathbf{X}(t_0) = i] = P[\mathbf{X}(t_0) = x_0]\mathbf{V}_k(t, t_0) \tag{9.12}$$

where $V(t, t_0), t > t_0$ is the N by N matrix of the conditional probabilities, with $V(t, t) = I$, the identity matrix and $p_{ik}(t, t_0)$ is the probability of moving from state i to state k in the time $t - t_0$. The probability distribution in X at time t can be derived from (9.12):

$$\frac{dP(t)}{dt} = P(t)Q(t) \tag{9.13}$$

where $\sum_{k \in X} P_k(t) = 1$

$Q(t)$ is the matrix of the transition rates, and is defined as:

$$Q(t) = \lim_{\delta t \to 0} \frac{V(t + \delta t, t) - I}{\delta t} \tag{9.14}$$

9.3 Existing techniques for designing systems to be reliable

Redundancy is the principal tool engineers use to combat failure. A number of different forms of redundancy exist.

9.3.1 Static redundancy

Hardware redundancy is the use of additional resources to reduce the reliance on single components. In its simplest form this is static or N-modular redundancy (see figure 9.11): duplication of particular modules N times so that if $N-1$ modules fail, the entire system will still operate. Often a module failure is only detectable by comparing its result to that of a number of other identical modules via a voting system. This reduces the effective redundancy to a system failure in the event of $\frac{N}{2}-1$ modules failing.

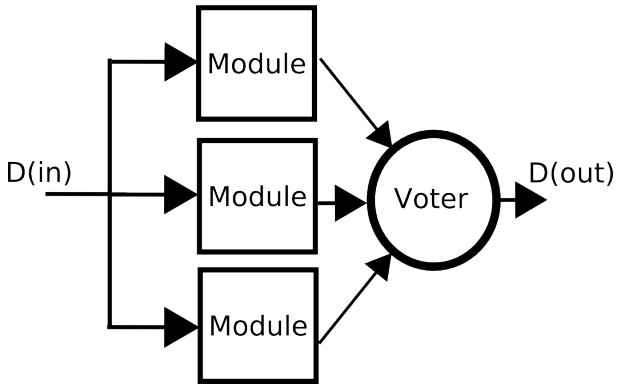


Figure 9.11: An example of static triple-modular redundancy

The reliability of a triple-modular redundant system,R_{3M}, is the probability of three modules operating correctly plus the probability of any two modules operating correctly (see equations (9.15) and (9.16)). If you consider the reliability of the voter, R_V, the reliability equation becomes (9.17). Equation (9.18) is the reliability of an N-modular redundant system.

$$R_{3M} = R_V(R_M^3 + 3R_M^2(1 - R_M)) \tag{9.15}$$

$$R_{3M} = R_V(3R_M^2 - 2R_M^3) \tag{9.16}$$

$$R_{3M} = R_V(3R_M^2 - 2R_M^3) \tag{9.17}$$

$$R_{NM} = R_V \sum_{i=0}^{(N-1)/2} \frac{N!}{i!(N-i)!}(1 - R_M)^i R_m^{N-i} \tag{9.18}$$

After the failure of one module, the reliability of a TMR system is less than that of a single module (see figure 9.12). This is because a failure in either of the two remaining modules or the voter will now cause a system failure. Figure 9.13 shows the effect increasing the available redundancy has on the reliability of the system.

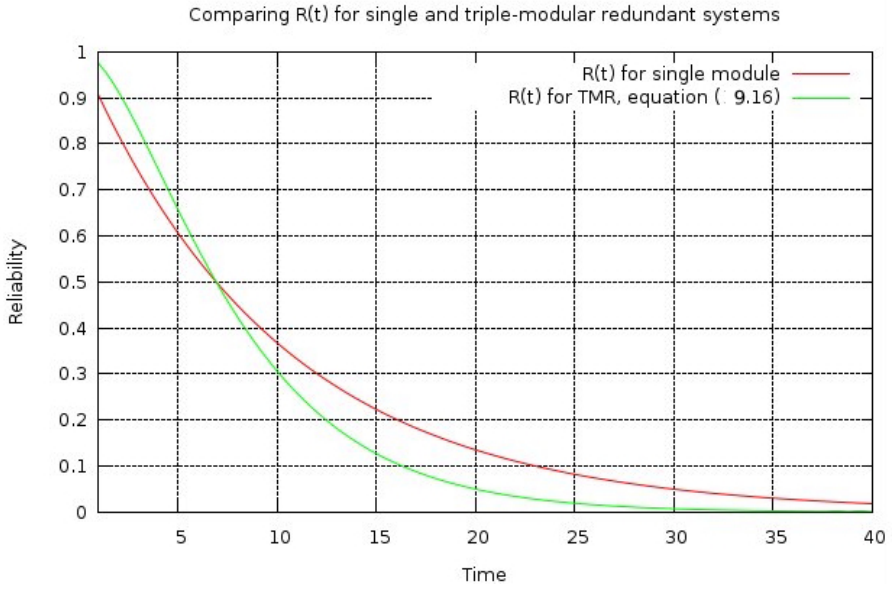


Figure 9.12: A comparison of the reliability of single and triple-modular redundant systems

9.3.2 Dynamic redundancy

A more flexible, usually more efficient, reliability scheme is dynamic redundancy. Dynamic redundancy uses a two-step procedure for the elimination of a fault. First the presence of a fault is detected, second, appropriate corrective action is taken.

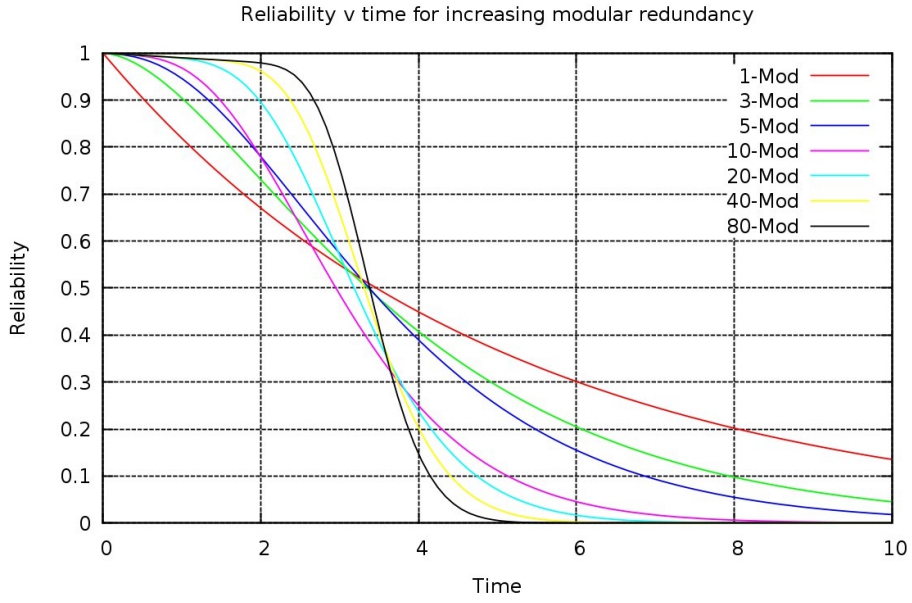


Figure 9.13: A comparison of the reliability of increasing modular redundant systems

A good example is an offline uninterruptible power system (UPS) which activates a redundant power supply in the event that a failure of the primary power supply is detected. This subsection describes a small selection of the many different types of dynamic redundancy:

1. **Information redundancy**: Checksums and parity bits are examples of information redundancy. These are extra bits used to confirm that the received or retrieved data is the same as that which was previously sent or stored and thus eliminate any errors caused by transient faults.

2. **Time redundancy**: The watchdog timer is a timeout mechanism that restarts a process in the event that it fails to complete its responsibilities within a specified period.

3. **Audits**: This form of redundancy requires a detailed knowledge of the system to which it is applied. An audit is an algorithm that checks the consistency of different data structures. The Xenon switching system, used in modern telephone exchanges, has a good example of a periodic audit. As part of the system audit test, a check to compare the call occupancy of the system

and the number of failed calls is called. If the call occupancy is much less than maximum and calls are still failing, a periodic audit will detect this and trigger another audit. This audit will cross-check the processor responsible for allocating calls to routes and the processor responsible for requesting call routes in the first place.

4. **Task rollback**: Although difficult to implement in hardware, the principle of task rollback is simple: if the device has entered an erroneous state then the system is reset to the last correct state. In practice this means storing quantities of data that describe the state of components at specific checkpoints within an algorithm.

In the next chapter we will use the reliability analysis techniques: minimal cut sets, fault tree analysis, failure mode effects and criticality analysis and a parts count to determine the reliability of a morphogenesis-inspired reliable ALU and an equivalent triple-modular redundant system.

Chapter 10

Morphogenesis-inspired ultra reliability

In previous chapters we have discovered and imitated the ability of biological systems to create self-assembling patterns. If, instead of mapping to an output colour, the state of each cell maps (many-to-one) to a component type within an electronic system, the system will self-assemble into a potentially functional circuit [MB03].

A similar technique was proposed and an "Embryonics" frequency divider was implemented by Cesar Ortega-Sanchez *et. al.* [CMST00]. This used an array of identical cells that determine their function with a cartesian co-ordinate system that uniquely labels each cell. Each label corresponded to a cartesian genome (see chapter 3) that described the function assigned to that cell. The work presented in previous chapters allows us to use fewer labels and function descriptors, but at a cost of a more complicated system and design algorithm.

10.1 Complexity versus reliability

To be of any use, a reliability mechanism must be less prone to failure than that which it is trying to protect. However, as each cell must be identical, achieving this homogeneity for large system blocks adds a significant cost in replicated function to the design. Thus the optimum scale at which to discretise the design into a cellular architecture is found at the minima of total cost of system hardware (see Figure 10.1).

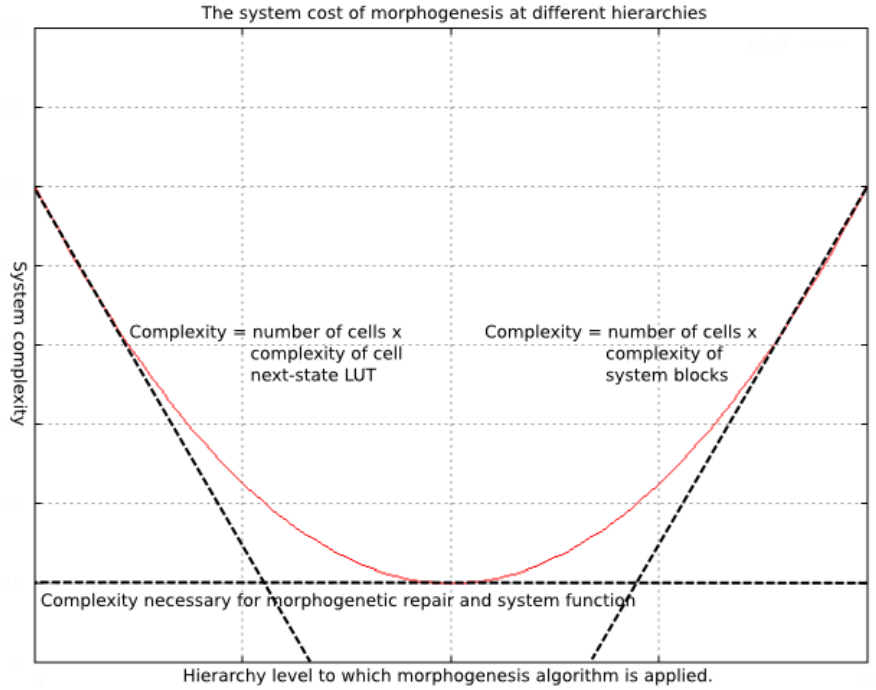


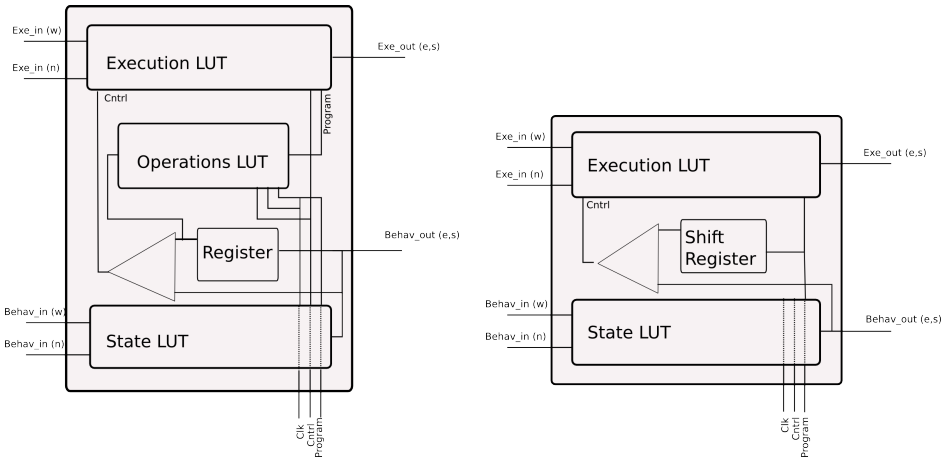
Figure 10.1: The system cost of morphogenesis at different hierarchies

10.2 ASIC implementations

Existing reprogrammable devices (PLA, PAL, FPGA) are not optimised for the fine-grained self-reprogrammable logic required for this self-assembling algorithm. An appropriate custom Application Specific Integrated Circuit (ASIC) could be optimised to use fewer units of logic than an equivalent FPGA implementation. One possible embodiment of a cell within an ASIC is shown in figure 10.2(a). The state

of each cell must map to a component function, coded in the form of a bitstream that can be written to a look-up table and executed. Every bitstream required for the automata is stored in the function look-up table. The bitstream is selected by the cell state, loaded into the execution look-up table and executed. Every time the cell configuration changes (as detected by the cell comparator), the execution look-up table is reloaded with a new bitstream.

An alternative embodiment is shown in figure 10.2(b). The function LUT is replaced with much simpler logic and a sequence comparator. An externally stored program is perpetually transmitted to each cell serially via the "program" line. Each bitstream required for the automata is preceeded by a header that corresponds with the two input-states to the cell. Each cell compares its inputs to those in the headers and, if they match, re-programs the execution LUT with the bitstream that follows it.



(a) (b)

Figure 10.2: Alternative ASIC implementations

Another consideration is the use of redundant cells. Biological implementations of morphogenesis have an advantage over any electronic implementations: in the event of a cell being permanently damaged, biology can grow a replacement. This is something that is currently not possible in electronic devices. Thus, redundant cells — cells that can take the place of any other in the event of permanent failure — must be an integral part of the design. In order for a redundant cell to take the place of any damaged cell, every cell would have to be directly connected to every

other cell. An alternative is to place every cell on a shared bus and provide each cell with an appropriate interface. Dynamic, in-situ re-programmable routing is another possibility. There are already various algorithms [TSAT03] for managing dynamic routing.

A final possibility is some sort of fail-safe strategy for each cell, limited re-routing capabilities designed and programmed *a priori*, and strategically placed redundant cells. One implementation is the application of multiplexers that respond to any given permanent cell failure with a lateral displacement of inputs and outputs of the cells on its row, as seen in figure 10.3. This ensures that the automata platform can replace one cell in every row with an unused cell on the same row. In the event that more than one cell has failed on the same row, requiring an impossible lateral displacement of two cells, the multiplexers of the entire row will respond with a vertical displacement of inputs and outputs. This ensures that an automata platform can replace entire rows of faulty cells with unused rows of cells on the automata.

10.3 A self-assembling self-repairing one-bit full-adder

Let us use these morphogenesis-inspired principles to design a robust one-bit full adder. The schematic for a one-bit full-adder can be seen in figure 10.4

10.3.1 Design considerations

In order to minimise the component count (and therefore the number of components that contribute to the device failure rate) there are a few constraints to the schematic design:

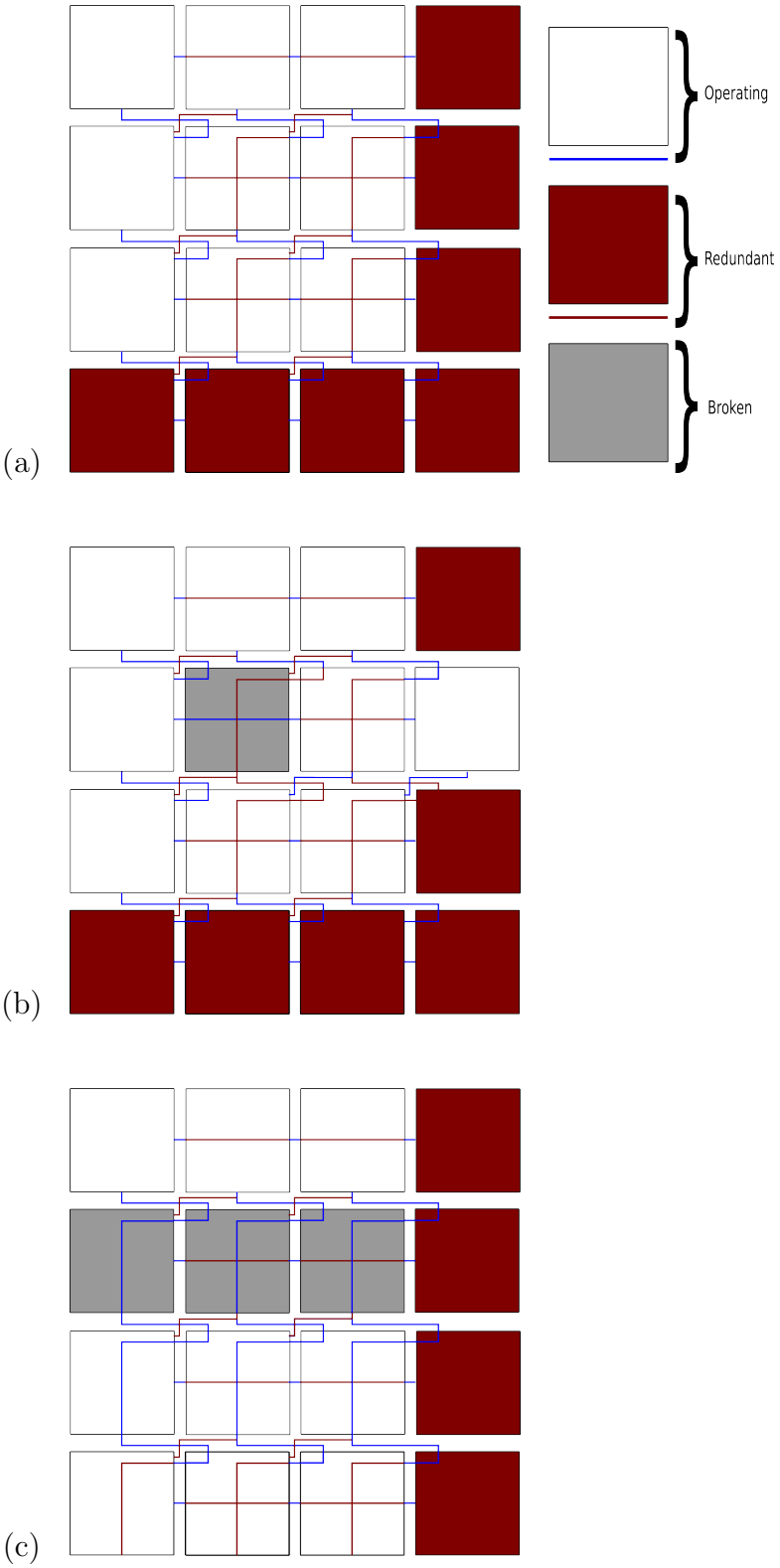


Figure 10.3: A fail-safe re-routing algorithm. (a) No broken cells, (b) one broken cell, (c) three broken cells

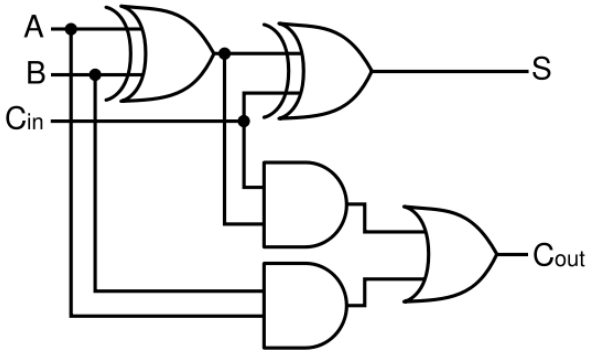


Figure 10.4: 1-bit full-adder schematic

1. Every cell determines its state from two of its immediate neighbours. Other than power and clock lines there are no global connections required for the CA to converge to its correct state. If a full-adder is to be implemented on such a platform, it would ideally not require any global connections either.

2. The cell has no bi-directional communications: it relies on two-inputs and two-output lines in a feed-forward arrangement in order to converge. Likewise an ideal full-adder design should be built on this arrangement. This means each component cell cannot feed back data to a cell that lies earlier on on the data path.

3. Each cell has two output lines, but the state-output is common to both. Again, the most appropriate implementation of a full-adder design will piggy-back these existing communications lines and not require additional networking. One consequence of this is that no two data lines can cross.

4. Because we want this full-adder to be scalable, the one-bit full-adder modules should be stackable, that is, if the modules are arranged one on top of one another, the carry-out lines should connect to the carry-in line beneath it.

5. In order for the full-adder to be scalable, the CA state pattern must repeat until it uses all the available cells.

6. There should be as few different cell-types (equivalent to the size of the CA alphabet) as is necessary, and there should be as few cells per one-bit module

as is necessary.

Figure 10.5 shows how the schematic of figure 10.4 has been revised to ensure there are no crossed data lines. Figure 10.6 shows the different cell operations and their corresponding state assignments. Figure 10.7 shows the schematic laid out over 16 cells. Note that this design requires the following boundary conditions:

1. The cell connected to input "A" of the full-adder must be to the right of a state "7".
2. The cell connected to input "B" of the full-adder must be to the right of a state "2".
3. The top-left cell of the first bit of the full-adder must be below a state "1".
4. The cell to the right of the top-left cell of the first bit must be below a state "2". Since the bottom row of each 16 cell design starts with the states "1" and "2", these boundary conditions propagate to the subsequent bits and the design repeats until it runs out of cells.

Listing 10.1 shows the pseudo-code of a VHDL implementation, the complete listing can be found in appendix A.3. Figure 10.8 shows the design (implemented on an ALTERA FPGA) self-assembling.

```
1  entity cell
2          has inputs: a_in (data), b_in (data), n_in (state), w_in (state)
3          has outputs: c_out (data), d_out (data), e_out (state), s_out (state)
4
5  architecture of cell
6          has a shared variable: state
7          has a ROM array: state_table
8                  (0,1,1),
9                                  ...
10                 (7,3,4),
11
12         has a process: determine_next_state:
13                                 loop entries in state_table till first two entries
                                       match (n_in, w_in)
14                                         next_state <= the third entry in the
                                               state_table
15
16                                 s_out, e_out <= next_state
17
```

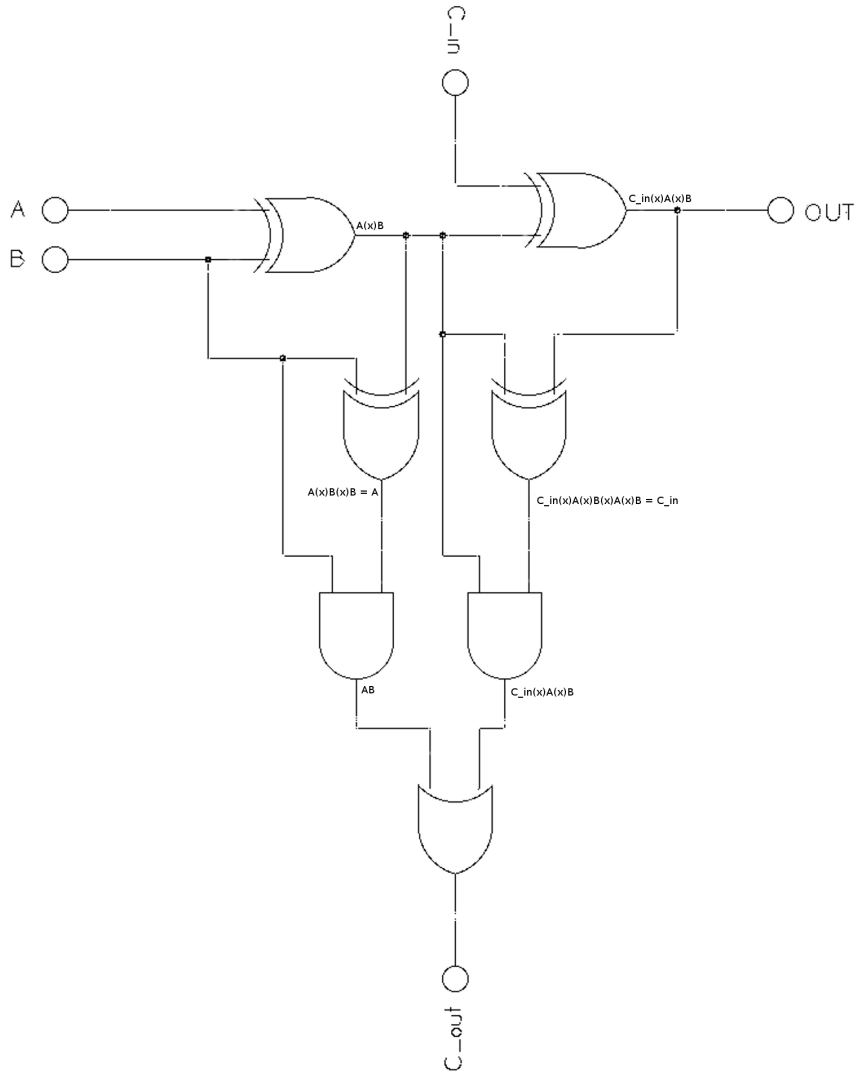


Figure 10.5: Alternative one-bit full-adder schematic

Cell function	OR		XOR		XOR/ AND
Cell states	0	1,7	2	3,4	5

Figure 10.6: Different types of cell

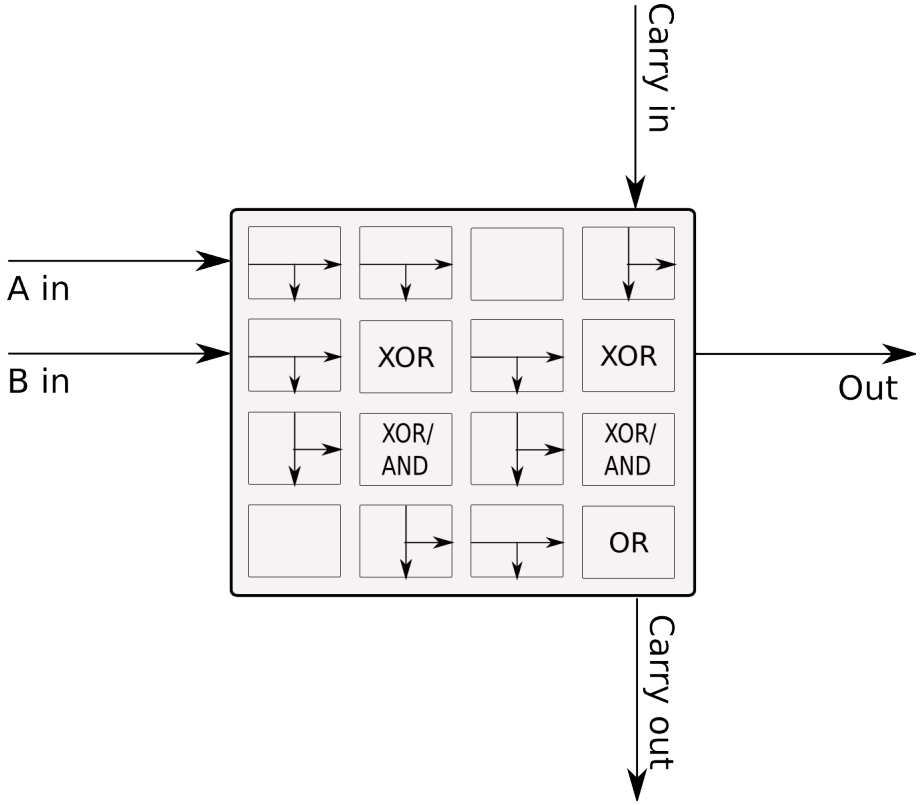


Figure 10.7: 1-bit full-adder layout

```
18        has a process: arithmetic
19                                case state is
20                                                when 0 => ans := b_in;
21                                                when 1 => ans := a_in or b_in;
22                                                ....
23                                                when 7 => ans := a_in;
24                                end case;
25                                c_out <= ans;
26                                d_out <= ans;
```

Listing 10.1: Full-adder pseudo-code

10.3.2 A self-reconfiguring ALU design

In order for the full-adder design to correctly self-assemble, the boundary conditions of the array must be precisely set. If these are changed, the arrangement of cell types will change. This effect can be taken advantage of by designing the cell array to respond to changes in the boundary conditions with desired alternative arrangements. Thus, the full-adder could be converted into a full-subtracter by

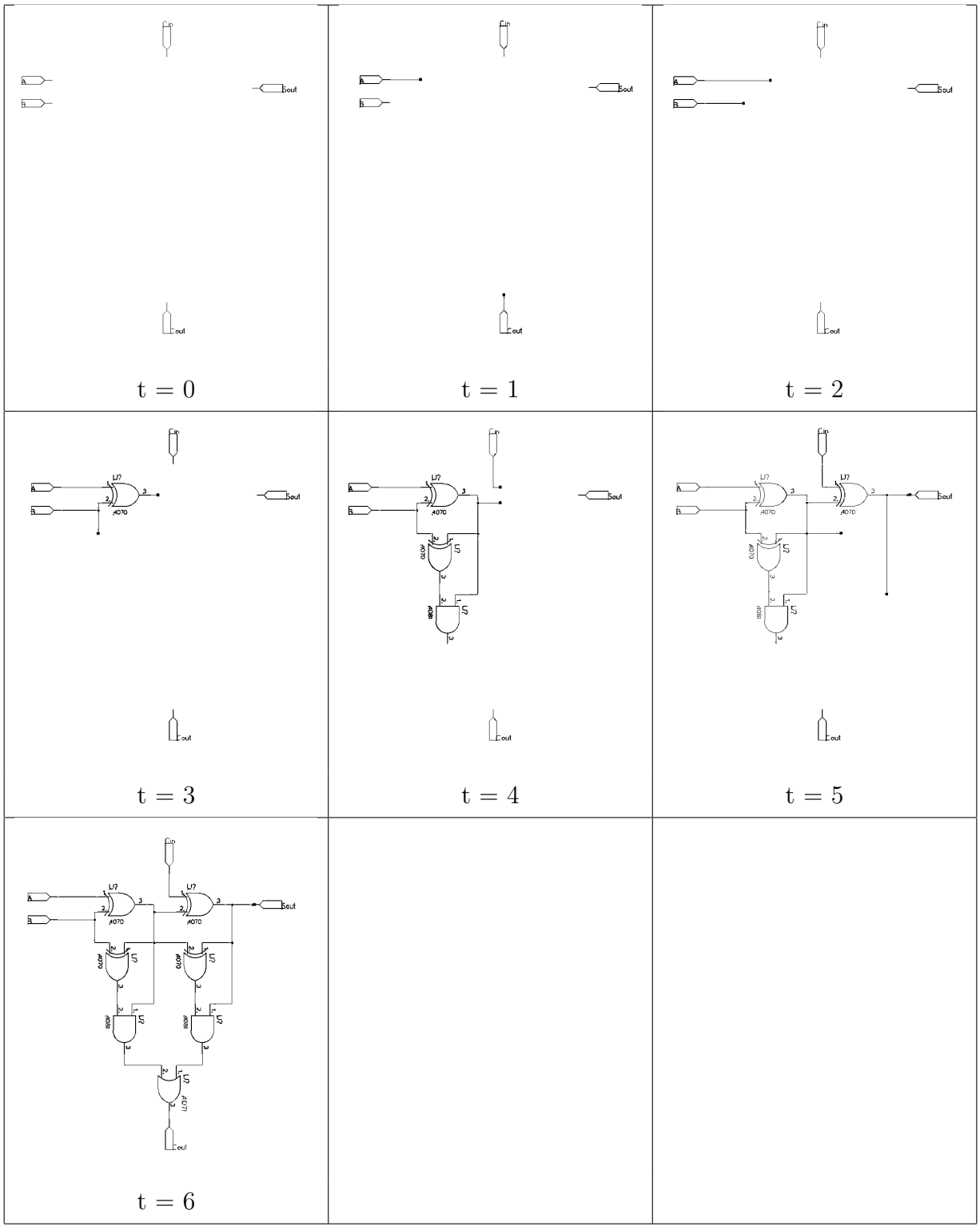


Figure 10.8: A 1-bit adder self-assembling

changing one of the boundary conditions. Likewise, the array can be programmed to perform other functions of an arithmetic logic unit (ALU) (e.g. AND, OR and NOT gates) with different boundary conditions.

Since we want the design to scale, these boundary condition changes need to propagate to the bottom of the 16 cell arrangement so that the subsequent 16 cells can also re-configure to perform the requested function. This requirement is responsible for some of the more esoteric logic arrangements (for instance a NOT gate being built from two XOR gates and a NOT gate) present in the designs. Figure 10.9 shows the logic arrangements and boundary conditions for the ALU functions, AND, OR, NOT and SUBTRACT.

Figures 10.10 — 10.13 shows the AND, OR, NOT and full-subtractor designs self-assembling.

10.4 Noticing failure

It is not enough for a system to be able to self-reconfigure and self-repair: it also needs to know if it is broken in the first place. A logical extension to this bio-inspired study is to imitate a biological "built-in self-test" (BIST) mechanism. When a biological cell suffers a hard failure, an internal BIST triggers "Apoptosis", a biochemically and morphogenically distinct form of cell death. If necessary a new cell is then formed to take its place. One such BIST trigger is the Cytochrome *c* chain reaction that is caused by DNA corruption[LKY+96].

Cytochrome *c* is a small protein associated with the inner membrane of mitochondria. Cytochrome *c* is released by the mitochondria in response to DNA corruption. This is preceded by a sustained elevation in the cell calcium levels. Cytochrome *c* interacts with the endoplasmic reticulum (ER) to prevent calcium inhibition of ER calcium release. The increase in calcium levels in turn triggers an increase in the release of cytochrome *c* which then acts in the positive feedback loop to maintain ER

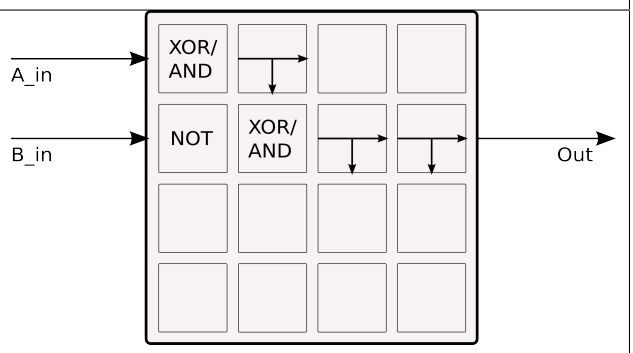
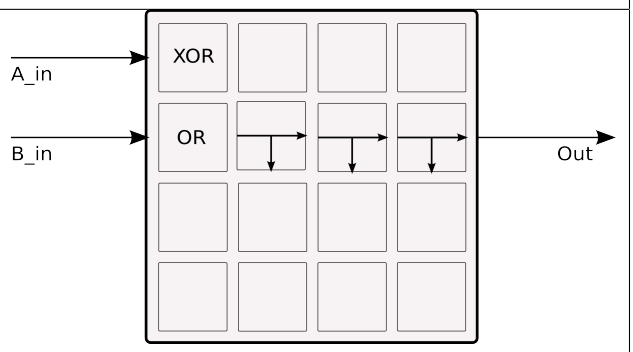
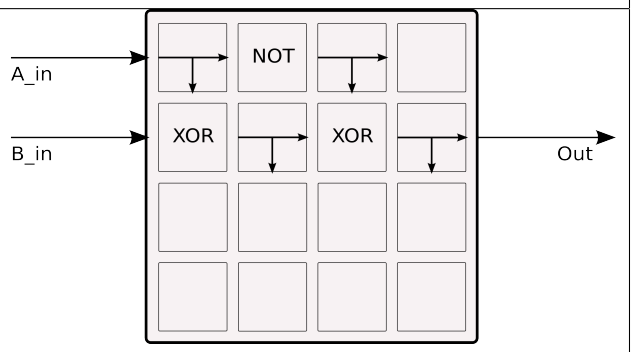
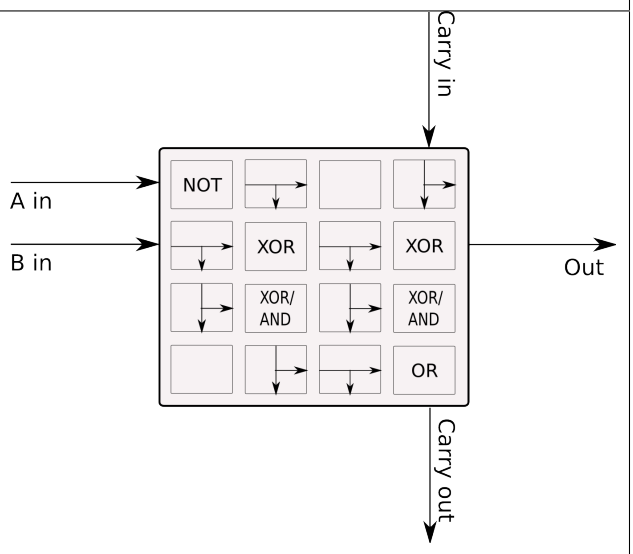
Function and boundary conditions	Cell arrangement
AND, (2,7)	
OR, (7,7)	
NOT, (5,7)	
Full-subtract, (4,7)	

Figure 10.9: Cell arrangements and boundary conditions for ALU

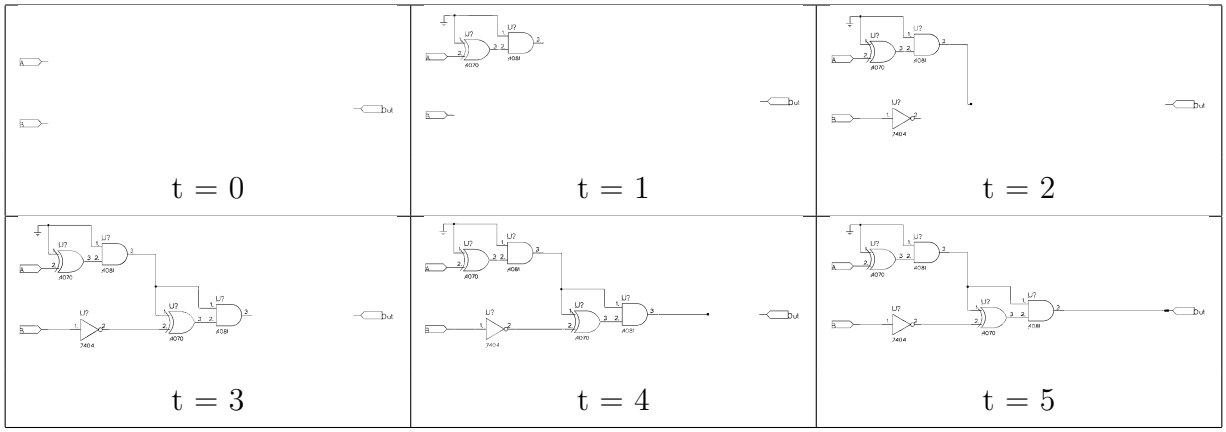


Figure 10.10: A 1-bit AND gate self-assembling

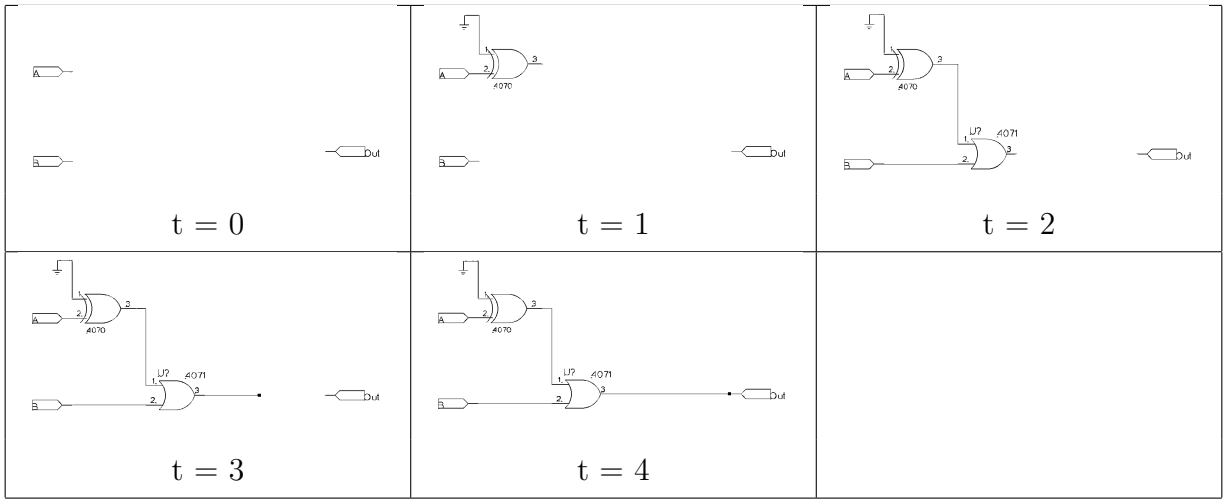


Figure 10.11: A 1-bit OR gate self-assembling

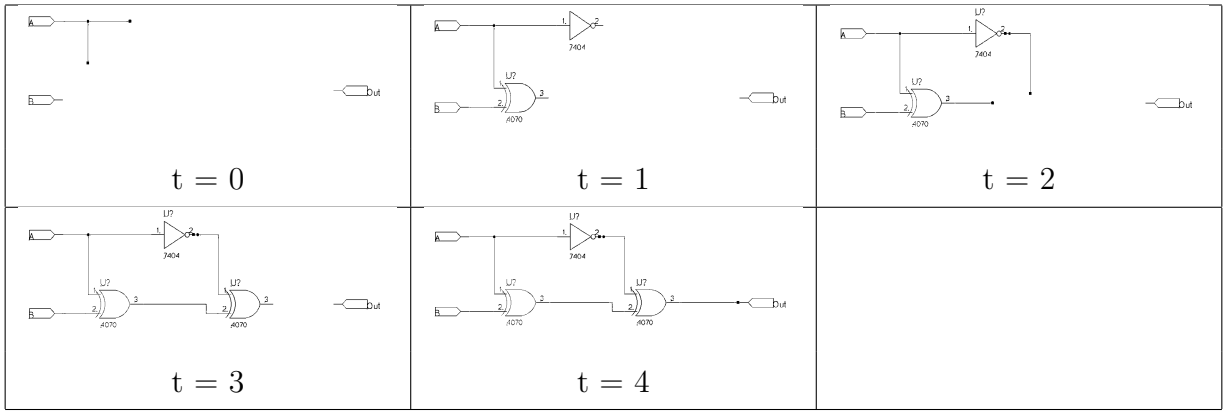


Figure 10.12: A 1-bit NOT gate self-assembling

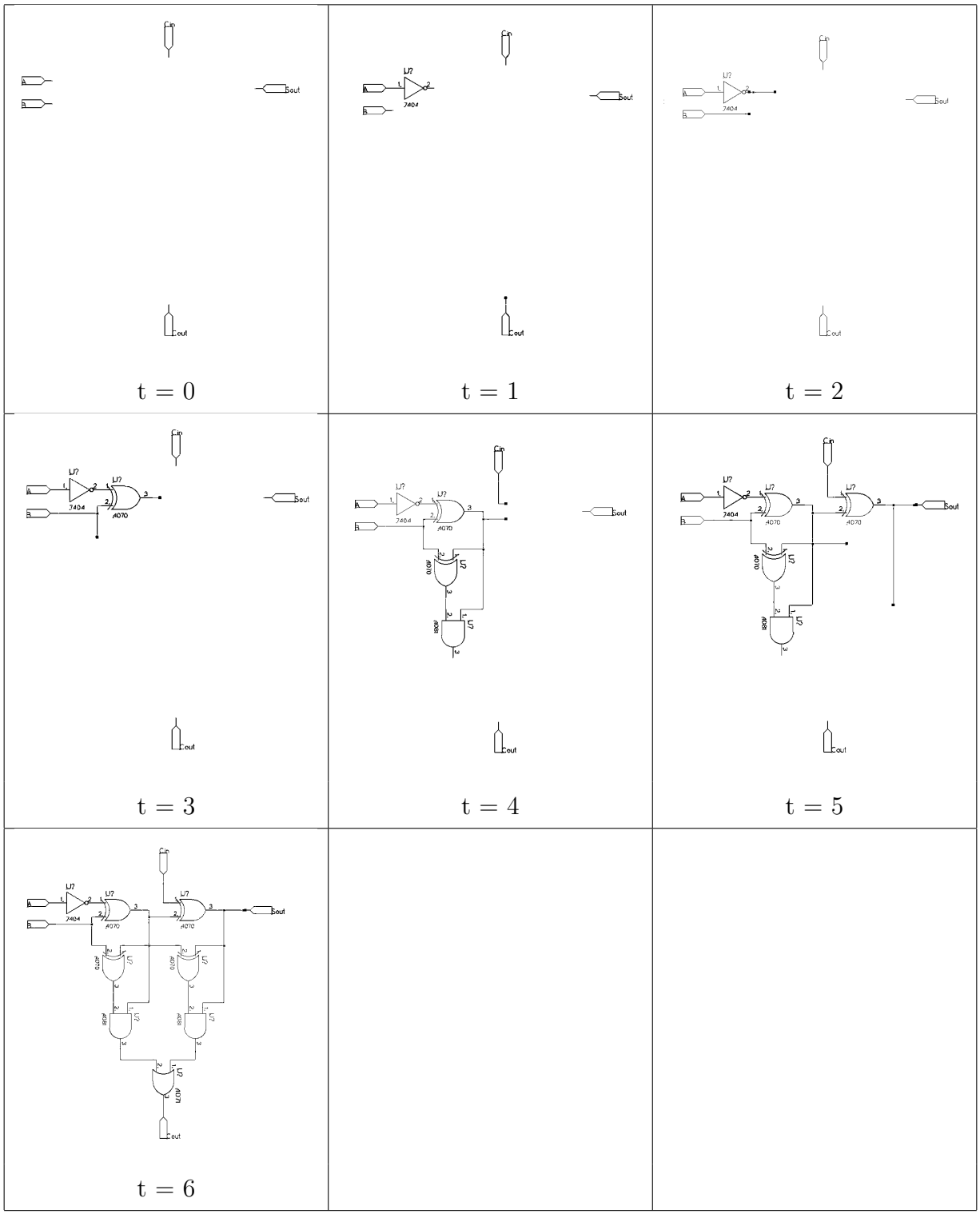


Figure 10.13: A 1-bit subtractor self-assembling

calcium release at pathologic levels. The calcium release in turn activates caspase 9.

Caspases are cysteine proteases, enzymes that cleave other proteins, and leave behind a cysteine residue that cleaves other proteins at the aspartic acid residue. Caspases are synthesised as inactive pro-caspases.

Caspace 9 then activates caspases 3 and 7, which are responsible for destroying the cell. [LKY+96] Apoptosis-associated nuclear condensation is usually accompanied by the activation of nucleases that first degrade chromosomal DNA into large subunits and then into smaller units of base pairs. [Wyl80] Plasma membrane integrity is maintained during apoptosis, which prevents the leakage of cytosolic contents into the extracellular domain. [RO00].

For the morphogenesis-inspired repair mechanism to be effective it is necessary to imitate, or provide an alternative to, this mechanism. For the purposes of this design we will assume hard failures can be modelled by "stuck-at" faults. If the data and state variables are Manchester encoded as it enters the ALU, the cell BIST simply needs to detect when a signal ceases to alternate between "1" and "0". If there is some such disparity, the monitor circuits trigger the electronics equivalent of apoptosis, namely making this functional unit transparent to its neighbours using techniques shown in figure 10.3.

10.5 Assessing the reliability of the ALU

The mechanisms of CA and morphogenesis require a homogeneous array of "stem" cells that are capable of becoming any type of cell during system development and repair. The LUT of FPGAs are conceptually just such a cell, but while some FPGAs are capable of partitioning and reprogramming a small portion of their entire configuration, the bit-stream configuration data must be provided from a source external to the FPGA. This means that without a significant re-design of the FPGA

infrastructure, the application of mitosis (replication of cell function) is not possible. As a result it is no longer enough for each cell to be capable of becoming any other type of cell; instead, each cell must be capable of performing the function of any other type of cell without any in-situ re-programming.

In order to assess the reliability of this self-repairing ALU and compare it to the reliability of a standard ALU design and an N-modular redundant ALU design, each will be implemented on a field programmable gate array (FPGA). The number of logic blocks used by each component of each design and the systems dependence on each will be used with the reliability analysis techniques described in the previous chapter in order to assess the MTBF of both.

There are a number of assumptions in this analysis:

1. The FPGA is an Altera Stratix II, the design software is Quartus. The fitting process used by Quartus to convert the VHDL design into a netlist will not necessarily create a netlist that uses the fewest number of logic units. As the self-repairing ALU is more complicated than a standard ALU, if the conversion does not create the smallest design possible the self-repairing ALU will be affected to a greater extent. However, for this analysis it is necessary to assume they are both affected equally.

2. Each design scales linearly. Thus an eight-bit ALU will use eight times the number of logic units as a single-bit ALU. Also a triple-modular redundant system will use three times the number of logic units (plus more for the voter) as the single ALU.

3. The MTBF of the design can be assessed by considering the MTBF of the FPGA and multiplying by the fraction of it used by the design. While this ignores single points of failure (clocks, power lines, etc) it is not unreasonable as the most common failure-mode of an FPGA are temporary, localised degradation, typically due to stuck-at faults on interconnects [Tou99].

4. The ALTERA does not support partial reprogramming. Thus while the state

of each cell, their arrangement and interconnections are intrinsically self-repairing, there is no way for the function each cell performs to be corrected in the event of a failure. However, for the purposes of this analysis it will be assumed that the hardware does support reprogramming and the necessary logic is built into the BIST.

The VHDL code for the ALU design can be seen in appendix A.3. The cell component can be seen in figure 10.14 and a one-bit ALU comprised of 16 cells can be seen in figure 10.15

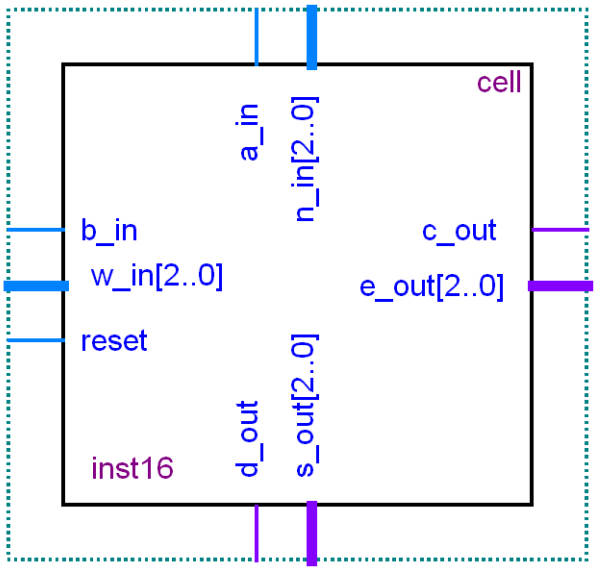


Figure 10.14: A cell of the ALU design

Below are the results of compiling these designs for the Stratix II FPGA.

A 171-modular redundant system uses the same resources as the self-assembling self-repairing design. A simple analysis shows the self-assembling solution can tolerate a greater number of failures than its static-redundancy equivalent.

The 171-modular system can tolerate $\frac{N}{2} - 1 = 85$ module failures without a system failure.

The self-assembling system consists of 128 identical cells. Provided the self-repair code is still working, the system can tolerate 127 cell failures at any one time because

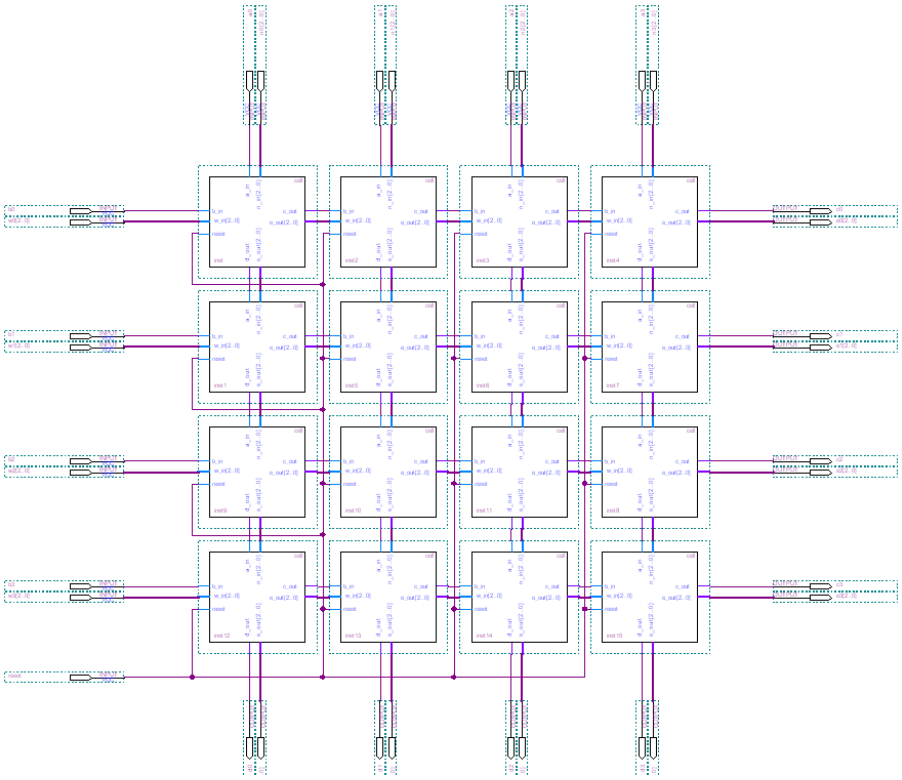


Figure 10.15: A 1-bit ALU made of 16 cells

Component	Logic units	% of FPGA
8-bit ALU	48	<1%
3-Mod 8-bit ALU	152	2%
— ALU	48	<1%
— Voter	8	<1%
171-Mod 8-bit ALU	9120	100%
8-bit self-assembling ALU	9120	100%
— Cell	87	1%
— BIST	1	<1%

Figure 10.16: Comparison of different ALU designs

each failed cell can copy the operating code from the one remaining cell.

Figure 10.17 shows the reliability parameters for equation (9.8) appropriate to the ALTERA Stratix II FPGA. Using the MIL-HDBK-217F standard and a part-stress analysis, the failure rate of the FPGA can be determined as $\lambda_p = 0.32$ failures per 10^6 hours.

Variable	Nomenclature	Category	Value
Base failure rate	C_1	PLA $>$ 5000 gates	0.042
Operating temperature	π_t	25 °C	0.1
Package failure rate	C_2	$>$ 300 Pins	0.16
Environment factor	π_E	Ground, Fixed	2
Quality factor	π_Q	Undocumented	1
Learning factor	π_L	$>$ 2 years	1

Figure 10.17: Reliability parameters for the ALTERA STRATIX II

While it is possible to form a fault-tree model of the ALU, this would be inappropriate because of its state-dependent failure modes. A more appropriate approach is the use of a Markov model (see figure 10.18) of the three operating states.

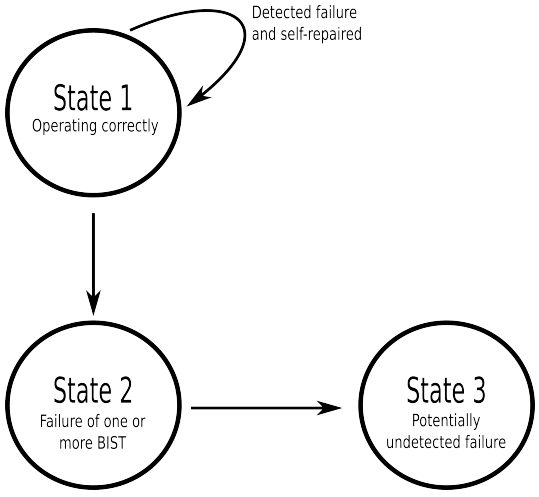


Figure 10.18: Markov model of the ALU, memory and processor systems

Using the Chapman-Kolmogorov equations (10.1) and (10.2), it is possible to analyse this model in order to determine the probabilities and dynamics of each failure mode.

$$\frac{dP(t)}{dt} = P(t)Q(t) \tag{10.1}$$

where:

$$P(0) = [1, 0, 0] \tag{10.2}$$

Q(t), the transition matrix, is derived from the Markov model. Each failure rate, λ refers to a transition between states.

$$Q(t) = \begin{pmatrix} -\lambda_1 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 \\ 0 & \lambda_2 & 0 \end{pmatrix}^T \tag{10.3}$$

From (10.1) and (10.3) it is possible to form three coupled differential equations, (10.4), (10.5) and (10.6).

$$\frac{d}{dt}P_1(t) = -\lambda_1 P_1(t) \tag{10.4}$$

$$\frac{d}{dt}P_2(t) = \lambda_1 P_1(t) - \lambda_2 P_2(t) \tag{10.5}$$

$$\frac{d}{dt}P_3(t) = \lambda_2 P_2(t) \tag{10.6}$$

The solution to (10.4) that meets the initial conditions (10.2), is (10.7).

$$P_1(t) = e^{-(\lambda_1 t)} \tag{10.7}$$

Equation (10.5) can be rearranged to form (10.8).

$$\frac{d}{dt}P_2(t) + \lambda_2 P_2(t) = \lambda_1 P_1(t) \tag{10.8}$$

Multiplying by the integrating factor $e^{\lambda_2 t}$, we obtain (10.9).

$$e^{\lambda_2 t}\frac{d}{dt}P_2(t) + e^{\lambda_2 t}\lambda_2 P_2(t) = e^{\lambda_2 t}\lambda_1 P_1(t) \quad (10.9)$$

$$\frac{d}{dt}[e^{\lambda_2 t}P_2(t)] = e^{\lambda_2 t}\lambda_1 P_1(t) \quad (10.10)$$

Substituting (10.7) into (10.10) gives (10.11).

$$\frac{d}{dt}[e^{\lambda_2 t}P_2(t)] = e^{\lambda_2 t}\lambda_1 e^{-\lambda_1 t} \quad (10.11)$$

$$\frac{d}{dt}[e^{\lambda_2 t}P_2(t)] = e^{(\lambda_2-\lambda_1)t}\lambda_1 \quad (10.12)$$

Integrating both sides with respect to t gives (10.13).

$$e^{\lambda_2 t}P_2(t) = \frac{\lambda_1}{\lambda_2-\lambda_1}e^{(\lambda_2-\lambda_1)t} + C \quad (10.13)$$

Dividing both sides of (10.13) by $e^{\lambda_2 t}$ gives (10.14)

$$P_2(t) = \frac{\lambda_1}{(\lambda_2-\lambda_1)e^{\lambda_2}}e^{(\lambda_2-\lambda_1)t} + \frac{C}{e^{\lambda_2}} \quad (10.14)$$

$$= \frac{\lambda_1}{\lambda_2-\lambda_1}e^{-\lambda_1 t} + Ce^{-\lambda_2 t} \quad (10.15)$$

Substituting the initial conditions (10.2) into (10.15) we can determine C.

$$0 = \frac{\lambda_1}{\lambda_2-\lambda_1}e^{(-\lambda_1)0} + Ce^{-\lambda_2 0} \quad (10.16)$$

$$0 = \frac{\lambda_1}{\lambda_2-\lambda_1} + C \quad (10.17)$$

Substituting (10.17) into (10.15) we get a final answer for $P_2(t)$.

$$P_2(t) = \frac{\lambda_1}{\lambda_2 - \lambda_1}(e^{-\lambda_1 t} - e^{-\lambda_2 t}) \tag{10.18}$$

The sum of the state probabilities must be one, because the system must reside in one of the three states. (10.19)

$$\sum_i P_i(t) = 1 \tag{10.19}$$

Thus the solution to (10.6) is found by subtracting the results from the previous two results. (10.20)

$$P_3(t) = 1 - P_1(t) - P_2(t) \tag{10.20}$$

Figure 10.19 shows the reliability $(P_1(t) + P_2(t))$ versus time of the self-repairing ALU and an N-modular redundant ALU of an equivalent size.

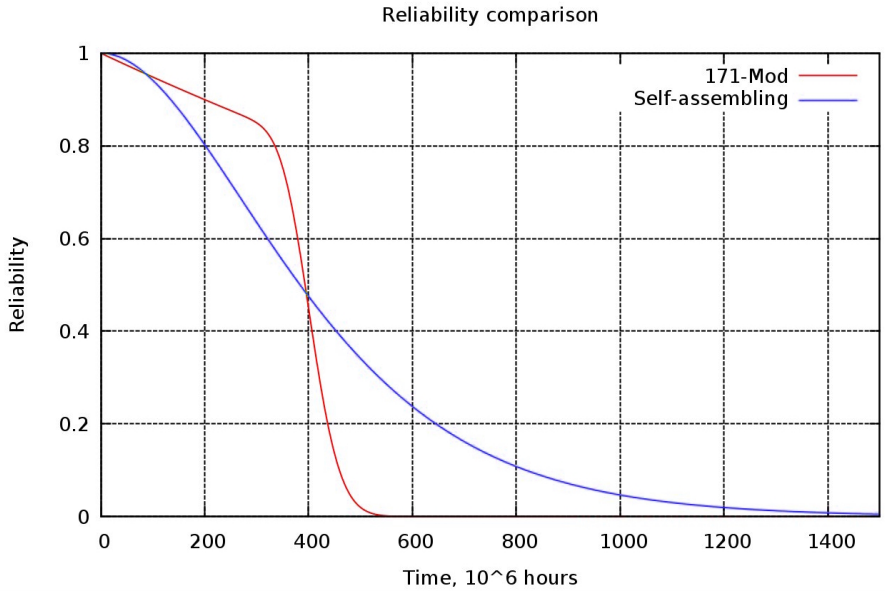


Figure 10.19: Reliability comparison (No redundant cells)

The reliability demonstrated by this self-repairing design is due to it being able to tolerate those failure modes that can be corrected by reconfiguring the device. The reliability of the self-repairing ALU can be further increased by introducing small numbers of redundant cells to the design to replace irreparably broken cells. Figure 10.20 shows the reliability of a self-repairing ALU with 1 redundant cell as well as the

reliability of an N-modular redundant system of equivalent size. Figure 10.21 shows the reliability of a system with 5 redundant cells, figure 10.22 shows the reliability of a system with 10 redundant cells.

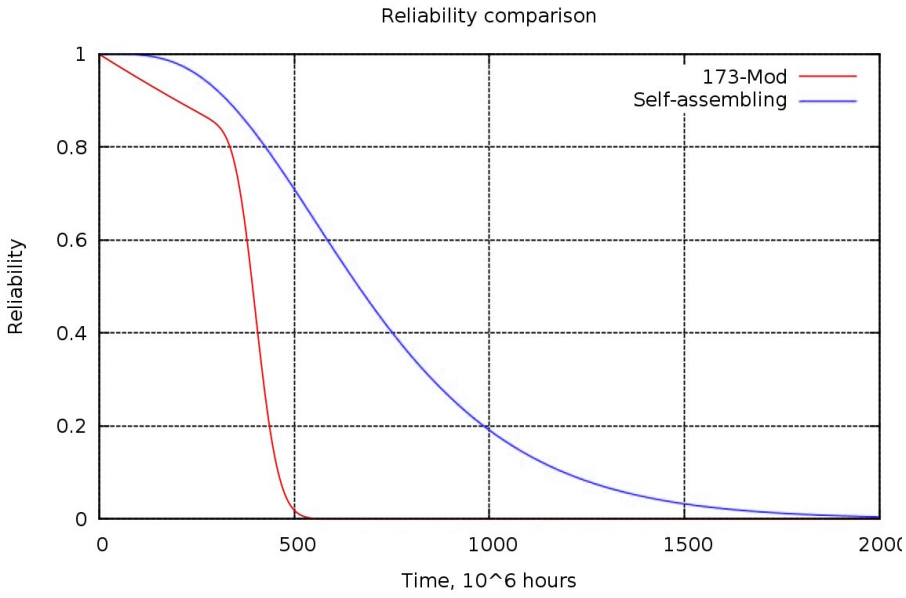


Figure 10.20: Reliability comparison (1 redundant cell)

In order to quantify the reliability improvement shown in figures 10.20—10.22 the MTTF of each system will be calculated at 400.10^6 hours using equation (10.21).

$$MTTF = \frac{400}{\ln\left(\frac{\lambda_1}{\lambda_2-\lambda_1}(e^{-400\lambda_1} - e^{-400\lambda_2}) + e^{-400\lambda_1}\right)} \tag{10.21}$$

With no redundant cells available to the self-assembling design, the MTTF is slightly longer than that of its N-modular equivalent. However making available small numbers of cells significantly increases its MTTF (by a factor of 10^{10} hours in the case of 10 redundant cells) performance over its equivalent N-modular design. This is subject to a number of caveats already discussed, as such this performance increase should be considered as an indicator of the potential for morphogenesis-inspired reliability. A more complete analysis would require detailed information of hardware overheads for each system component and a partial-reprogrammable platform with known failure modes.

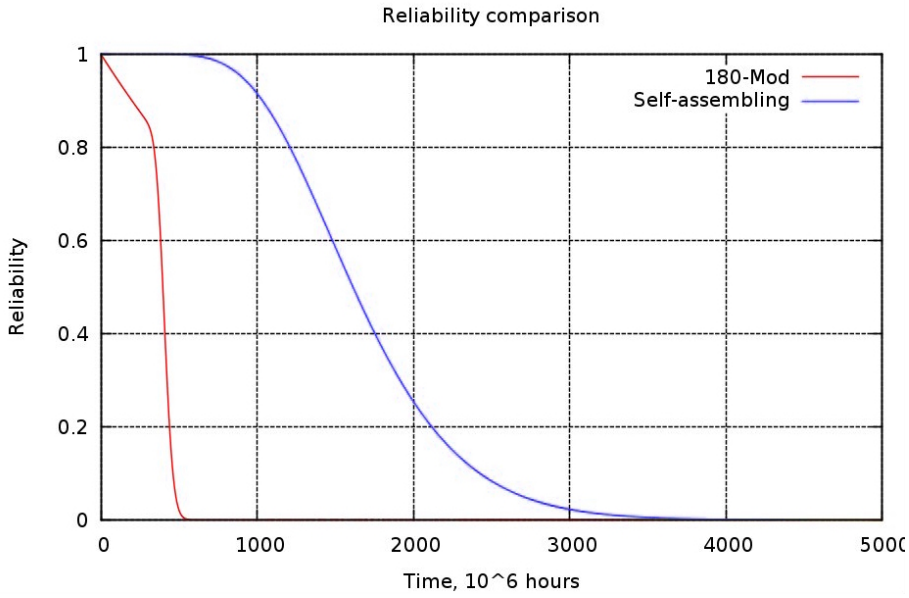


Figure 10.21: Reliability comparison (5 redundant cells)

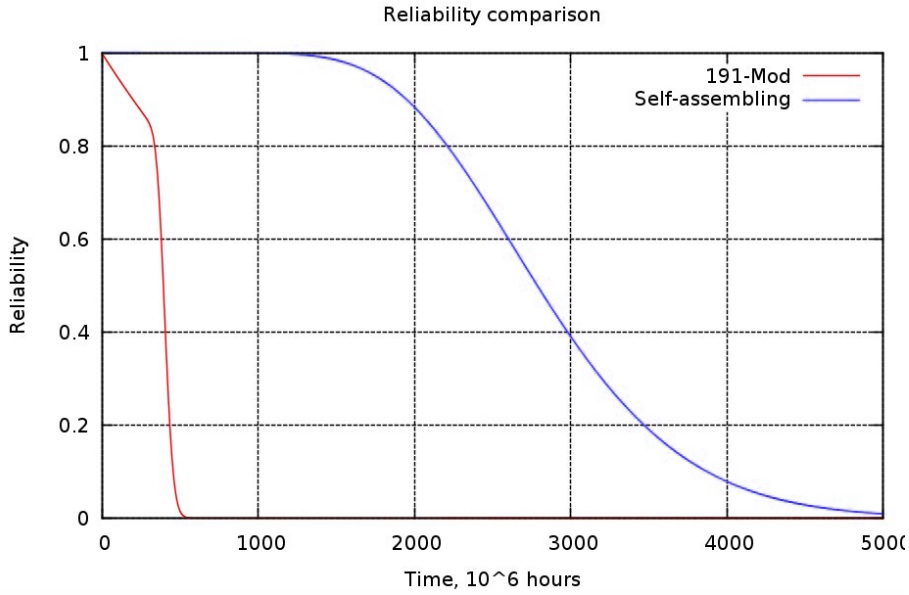


Figure 10.22: Reliability comparison (10 redundant cells)

Redundant cells	MTTF (10^6hrs) of self-assembling technique	MTTF (10^6hrs) of N-mod technique
0	$5.28.10^2$	$6.44.10^2$
1	$2.07.10^3$	$6.46.10^2$
5	$3.23.10^6$	$6.49.10^2$
10	$1.03.10^{12}$	$6.54.10^2$

Figure 10.23: Mean time to failure (MTTF) of ALU

10.5.1 Characterising the failure modes

How does this self-assembling strategy affect the "bath-tub" model for the lifetime of a system?

1. **Infant mortality**: If there exists redundant cells within the design, the system is much better equipped to deal with manufacturing defects. Thus the frequency of so-called "dead on arrival" products is potentially lower.

2. **Intrinsic failure**: Capable of self-repair, this system is almost completely impervious to soft errors. Only if every cell were corrupted simultaneously would the system fail during this stage of its lifetime.

3. **Wear-out failure**: The reliability assessment presented above is principally a measure of the resistance of the device to wear-out failures. It has been shown to potentially be a significant improvement on static redundancy techniques.

Using equation (9.2) and a Levenberg-Marquardt non-linear regression we can fit the previously calculated failure rates to a Weibull curve. Figures 10.24, 10.25, 10.26 and 10.27 show the hazard rates and their matched Weibull curve for four redundant cell configurations. Figure 10.28 shows the Weibull curve parameters for each.

Of note is the variation of the shape parameter. The closer the shape parameter is to one, the flatter the variance of the hazard rate with time. While this corresponds to a greater reliability, this is at a cost to the accuracy of any MTTF predictions.

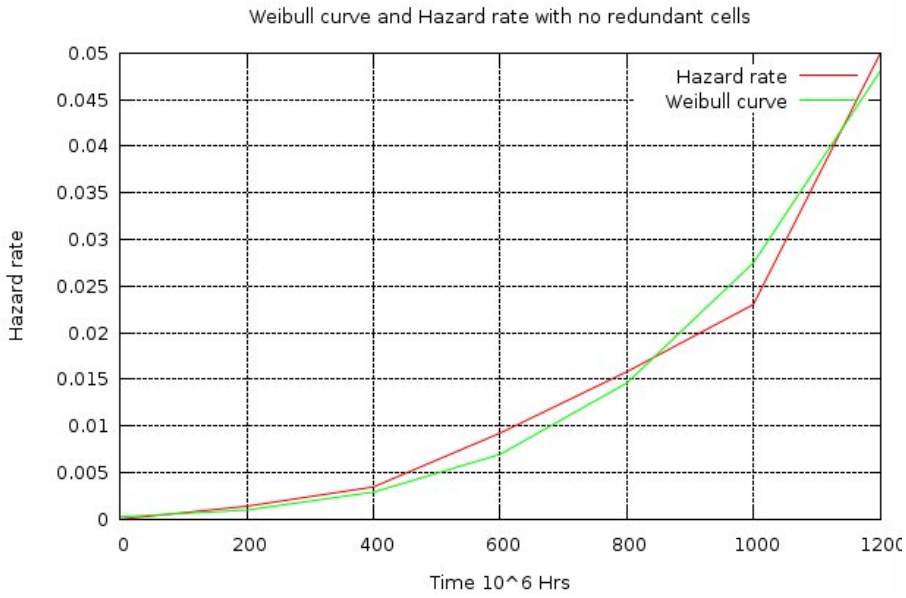


Figure 10.24: Weibull curves for self-assembling ALU

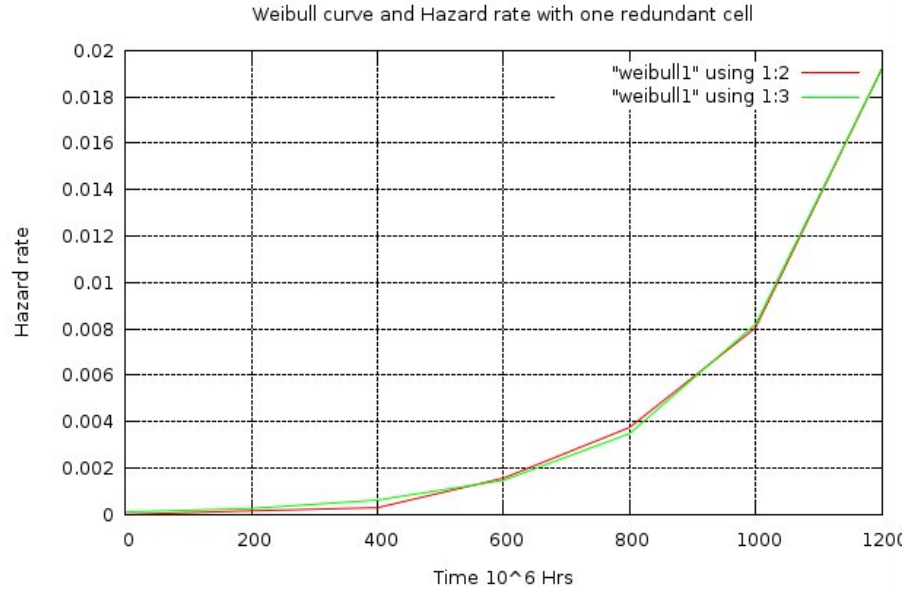


Figure 10.25: Weibull curves for self-assembling ALU with 1 redundant cell

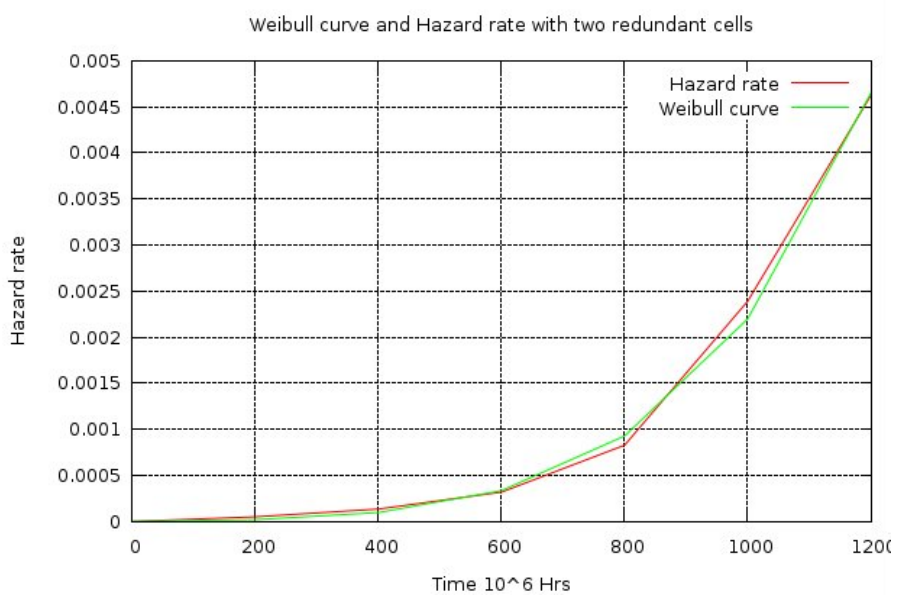


Figure 10.26: Weibull curves for self-assembling ALU with 2 redundant cells

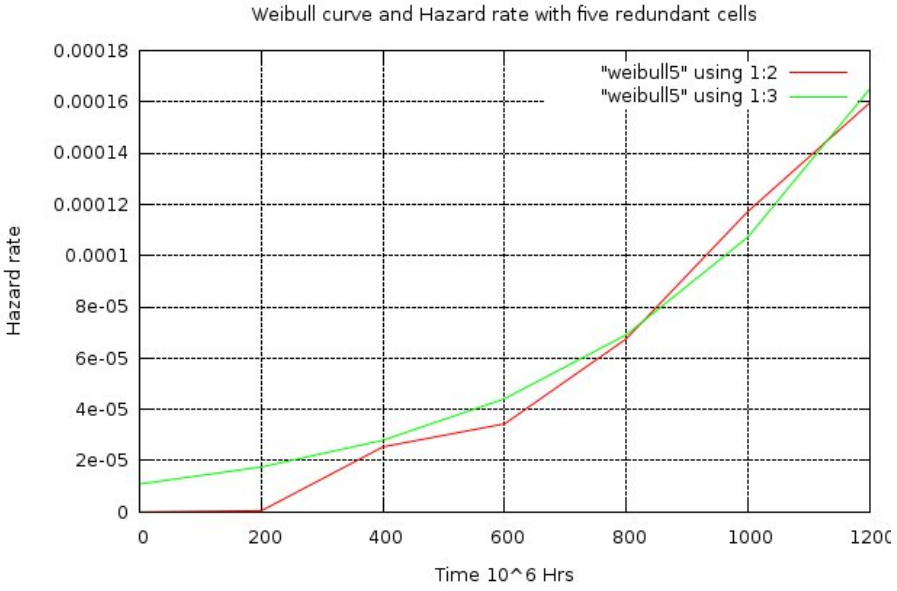


Figure 10.27: Weibull curves for self-assembling ALU with 5 redundant cells

Parameter	Redundant cells			
	0	1	2	5
Shape, β	79.9	120	56.2	32.8
Scale, η	$4.44.10^4$	$3.11.10^4$	$1.78.10^4$	$2.60.10^4$
Location, γ	$-3.76.10^4$	$-2.69.10^4$	$-1.28.10^4$	$-1.46.10^4$

Figure 10.28: Weibull parameters for self-assembling ALU

Often designers prefer to be able to accurately predict the MTTF and thus plan appropriate maintenance regimes.

10.6 Observations

Each of the 128 cells that make up the self-assembling ALU uses more logic units (87) than an entire standard ALU design (48). Thus by a simple parts count, the self-assembling design should have a failure rate at least 232 times greater than standard design. That this is not the case is, in part, because of the self-repair capabilities of the design — with no redundant cells this equates to a reliability that is comparable to that of an equivalent N-modular redundant ALU.

The addition of redundant cells to the self-assembling design reduces λ_2 according to N-modular redundancy equation (9.18). Furthermore, because one redundant cell can replace any of the 128 cells, its effect on the reliability of the entire system is approximately 128 times greater.

This technique, whilst demonstrating an improved reliability over an equivalent N-modular redundant system, is not without cost:

1. The minimum size of the self-assembling ALU is a factor of a hundred times greater than that of a single ALU and over thirty times that of a triple modular redundant ALU.

2. The design algorithm is not trivial. Modular redundancy can be easily applied to any system without a significant schematic re-design. This is not true of the self-assembling approach. Firstly an appropriate level for cellular discretisation must be selected. Next, the system function must be divided between cells such that short, preferably not overlapping, buses between nearest neighbour cells are the primary means of data transmission. Lastly the rules necessary to co-ordinate the assembly and differentiation of these must be generated.

3. The design is relatively simple — a four by four pattern repeated eight times. Increasing the complexity of the design will increase the number of rules each cell must obey, increase the number of gates required for each cell and thus reduce the reliability of the design.

For the reasons above, this approach may not be particularly applicable to the design of reliable, commercial projects. However it certainly fits into the "Ultra reliability" category. Any system that must operate in harsh environments (e.g. satellites and space shuttles), or that cannot tolerate failure, could benefit from this technique.

10.7 Conclusions

The self-assembling, self-repairing capabilities of morphogenesis can be mimicked in the design of self-assembling, self-repairing electronic circuits. These circuits are formed on an array of discrete, identical cells. To achieve correct system performance, each cell determines its state then uses this to determine its component type. Furthermore, these arrays can be designed to metamorphosise into different circuits depending on the boundary conditions of the array. This design approach has been demonstrated with an ALU design on an FPGA and its reliability assessed using state-based models and a stochastic analysis. Subsequently, this reliability was compared to the reliability of an N-modular ALU of equivalent FPGA logic unit use, and found to perform significantly better. A case for "Ultra reliable" applications was made, referring to the NASA program of the same name, and a study performed by the author for CERN.

Comparing this work to that of Cesar [CMST00] is not feasible because his "Embryonics" platform was used to perform a different function (a frequency divider) and was implemented on a different platform (the Xilinx Virtex XCV300 FPGA).

The designs shown in this chapter are currently the subject of patent (pending) GB0802245.1.

Chapter 11

Morphogenesis-inspired self-assembly

Self-assembly describes the spontaneous aggregation of pre-formed modules into well-defined, stable assemblies. This assembly happens without the assistance of any external co-ordinating processes.

The design of large self-assembling systems has been the bailiwick of chaos mathematicians and computational evolutionists for the last ten years because the relationship between locally-interacting modules and the general form of the larger assembly is not well understood.

To complicate matters further, a common assumption is that each pre-formed module is identical. This means that each component can take the place of another and the replacement of damaged blocks is trivial. Also, the challenge of self-replication is reduced from the procedure of copying the entire system once to that of copying a small component of the system many times.

The following are some of the reasons self-assembling systems are of interest:

1. **Robustness** The robustness of morphogenesis as a means of co-ordinating

assembly has been demonstrated in previous chapters. However in physical realisations of self-assembly the system robustness also benefits from the fact that these systems are usually composed of a large number of parts that can be interchanged and that can replace each other if one of them fails [KGPV08].

2. **Manufacturing** The possibility of bulk manufacturing elemental modules is attractive from a practical point of view as it cannot be expected that each component should be built independently. Bulk fabrication will ultimately make self-assembly an attractive concept for industry [WZBL05].

3. **Delivery** When the conduit cannot support a complete system, e.g. for reasons of weight or size, self-assembling sub-modules could be an effective alternative.

4. **Environmental** There is considerable interest in the potential, especially at micro and nano scales, of self-assembling systems. This is because there is a perceived economic and environmental advantage; self-assembly may be a more energy efficient, less wasteful alternative to existing manufacturing techniques.

Self-assembly has been studied in the context of biological systems, nanobots, chemistry and engineering. The IEEE [YWPS08] identified various grand challenges for robotics research in the 21st century. One of these was the demonstration of a self-assembling robot made of more than 1000 separate units. Specifically they recognised the need for a distributed algorithm to co-ordinate this self-assembly. This chapter will discuss an implementation of the convergent cellular automata algorithm presented in previous chapters to the design of large-scale self-assembling electronic robots.

11.1 Modular robotics

Robots are widely used in manufacturing, assembly and packing, transport, earth and space exploration, surgery, weaponry, laboratory research, safety and mass pro-

duction of consumer and industrial goods [oI08]. Modular robotics have potential advantages over their less dynamic counterparts: including ease of manufacture and repair, versatility and ease of transport.

The problem of delivering bulky medical devices to the stomach, be they for diagnosis or repair, has been the subject of a commercial modular robot developed by Given Imaging [MMSD07]. They have been testing designs for self-assembling, small, swallowable devices to create a surgical "robot". The work is interesting as a concept since modular robots assembling inside the body could enable more flexible and complex gastrointestinal-related robotic applications. Current capsules typically get bigger by adding more functionalities whereas self-assembling modules would have fewer such issues.

There is a growing interest in the design of self-assembling robotics. Figure 11.1 is a list of 16 ongoing projects and the degrees of freedom (DOF) of each module.

One example, the "Molecule" (see figure 11.2) is a robotic module capable of aggregating with other identical modules to form dynamic three-dimensional structures [KRVM98].

Molecules move by reconfiguring, or rotating about, their connections with other molecules. Figure 11.3 shows an aggregation of 4 molecules rotating and reconfiguring so as to move across the table.

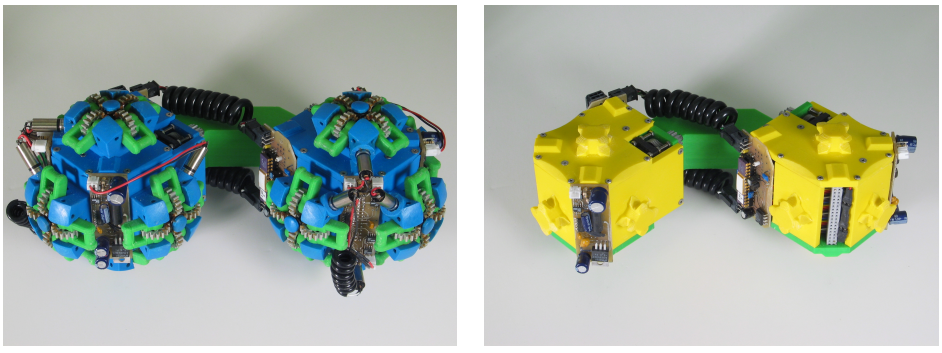
The requisite capabilities for each module of a self-assembling system are:

1. The ability to bond with other modules. The molecule project uses mechanical grippers, a self-assembling robotic chair [DDD08] uses a rack and pinion mechanism, the CKbot [YWPS08] uses electromagnets.

2. The ability to move relative to each other. This may be provided externally, for example by using a vibrating table or a cushion of air [GGJ05]. Alternatively, each module could move itself on wheels, by rolling itself over

Project	DOF	Homogenous	3D	Self-reconfiguring
ACM	1-3	Yes	Yes	No
Tetrobot	3-5	Yes	Yes	No
CEBOT	1-3	Yes	No	No
Fracta	12	Yes	Yes	Yes
Molecule	4	Yes	Yes	Yes
Metamorphic	3	Yes	No	Yes
Proteo	0	Yes	Yes	Yes
Crystalline	2	Yes	Yes	Yes
Fractal	6	Yes	Yes	Yes
Fractum	0	Yes	No	Yes
Mini unit	2	Yes	No	Yes
CONRO	2	Yes	Yes	Yes
I-Cubes	3	No	Yes	Yes
Polypod	2	No	Yes	No
PolyBot	1	Yes	Yes	No
SemiCylindrical	2	Yes	Yes	Yes

Figure 11.1: List of ongoing self-assembling robotics projects [JA01]



(a) (b)

Figure 11.2: Two types of Molecule module. The male (a) has an active gripper mechanism, the female (b) has a passive fixture.

Images courtesy of CSAIL, Cambridge MA

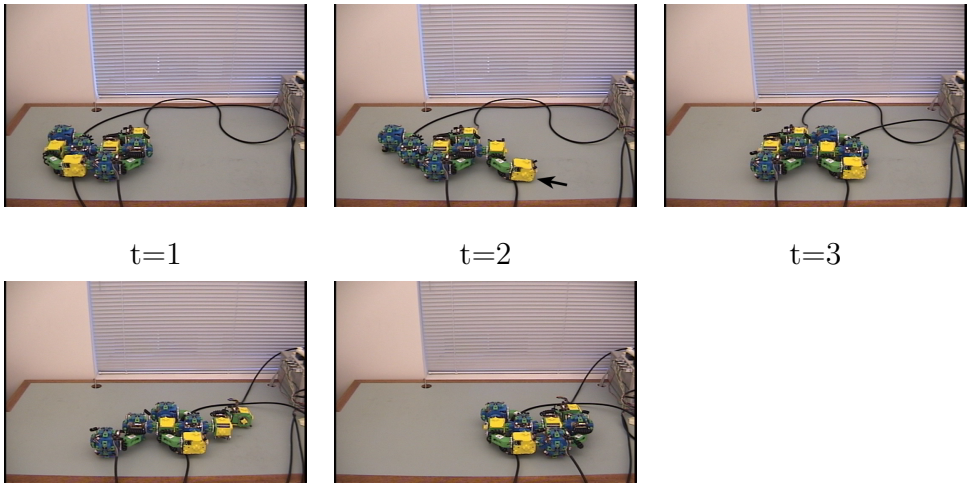


Figure 11.3: Five snapshots of a Molecule translation experiment.

Images courtesy of CSAIL, Cambridge MA

and over [YWPS08], or by pivoting about connections between other modules [KRVM98].

3. Some means of determining which modules it should connect to. Also, in more complicated systems, how and where it should connect to these modules. Current self-assembling robotics use a sufficiently small number of modules that complete Cartesian maps [BOT00] are more practical than the work presented in this thesis. However, as micro-electromechanical systems (MEMS) become more capable and cheaper to manufacture, a means of designing massive self-assembling systems may become more important. The following section will discuss a few other existing techniques.

11.2 Existing self-assembly techniques

There are many proposed schemes for the co-ordination of self-assembly. This section discusses three such schemes that have been designed for specific hardware implementations.

Hierarchical maps. Murata [MKTK99] proposed a hierarchical map for the self-assembly of mechanical structures from "fractum", small motorised robots capable of connecting to one another. The assembly program, stored in each cell, contains a description of the neighbourhood of every cell within a simple structure. Cells move randomly until every cell has a correct neighbourhood, at which point every cell freezes. Then another simple structure starts to form about one of the frozen cells. Because repeated simple structures need only be coded once, this algorithm is more efficient than a Cartesian co-ordinate mapping.

PacMan. Butler [BBR01] applied a two-stage process to the self-assembly of the "Crystaline" robot:

1. The difference between the current configuration and the required final configuration of cells is determined. A copy of the final configuration is passed as a map, one cell to the next, till it reaches the final cell. If a cell is required in a different location, it marks its current location on the map.

2. The route from each spare cell to its required location is determined. A recursive search starts, progressing one cell at a time in axis order depth, height, width, from the required location till a spare cell is found. Each step of the search is marked with a "plan-pellet". When a path is found, each "plan-pellet" en-route is turned into a "path-pellet". The spare cell will then move along this path, "eating" each path-pellet en-route.

Melt and Grow. Rus [RV01] proposed a different scheme for the Crystaline robot. The Melt and Grow assembly process is designed for systems of modules that can only move adjacent to other cells, so the system must remain a single contiguous cluster of cells. The initial configuration of cells is first "melted" into an intermediate structure, which is a projection of the robot modules into a pool on the ground. In a 2D world, this pool is a 1D line. When the pool is complete, the desired configuration "grows", one cell at a time starting from one end of the pool.

Overlapping circles. Nagpal [NKC03] described the desired final configuration using a network of covering circles. Overlapping circles are linked using local reference points relative to each circle. This circle representation permits the formation of the entire structure by agents recursively executing only two simple primitives: growing a circle, and triangulating the centres of adjacent circles.

Whilst appropriate to systems assembled from small numbers of complicated modules, the algorithms described above would not be trivial to implement on systems made of greater numbers of simpler modules.

So far the platform we have used for simulating morphogenesis has been a 2D rectangular cellular automata. Each cell is identical, each cell only communicates with its nearest neighbours and no co-ordinate scheme has been overlayed. On this platform, and without the assistance of a global supervisory algorithm, we have succeeded in designing the automata to converge to specified patterns.

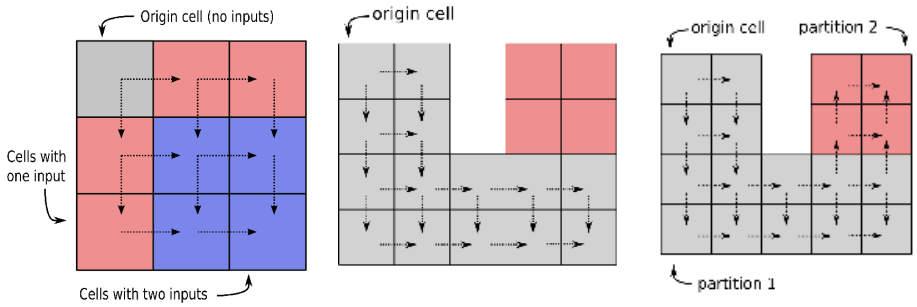
The design algorithm for this scheme is complicated and requires significant computational resources. However the assembly algorithm is local to each cell, trivial to implement and requires no long-distance communications.

There are no rectangles in biological systems; moreover, it is not normal to expect a self-assembling robot to form into a rectangle shape. The next section will adapt the current convergent cellular automata design algorithm to the more general case of self-assembling, irregular 3D systems capable of metamorphosis.

11.3 Design of irregular 2D automata

The scheme proposed so far relies on each cell computing its next state from the current state of two neighbouring cells; the relative location of each neighbour to the cell is common to every cell in the array. Therefore, if one cell updates according to the state of cells above and to the left of itself, so does every cell. In a rectangular

array of cells, most will have two neighbours which they determine their next state from, some will have one neighbour and one will have no neighbours. In effect the desired pattern emerges from this one cell (the origin cell): as shown in figure 11.4(a,b). In the case of a partially-assembled CA, or a CA that converges to form an irregular shape, this is not necessarily the case. This is shown in figure 11.4(c).



(a) (b) (c)

Figure 11.4: Inputs combinations, an irregular 2D CA and the flow of state information from the origin cell

A solution is to allow each cell to determine its next state according to the current state of two neighbouring cells as before, but to determine which two cells (above or below, to the left or right) according to its current state. Therefore, while one cell may determine its next state according to cells above and to the left of itself, another might determine its next state according to cells from below and to the right of itself. This "state to neighbourhood function" is a many-to-one mapping (see figure 11.5).

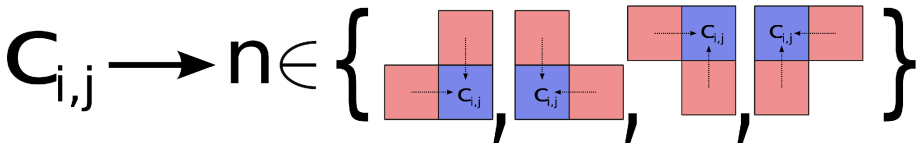


Figure 11.5: The state to neighbourhood function mapping

11.3.1 Analysis

If $g()$ is the two input one output state look-up table common to each cell, and $\overline{C_t}$ is the row-major vector of the cells in the automata, the transition from one state

to the next for an entire rectangular automata can be described as:

$$\overline{C_{t+1}} = g(\overline{C_t}) \tag{11.1}$$

Let $\overline{C_t} = \{\overline{P_1}, \overline{P_2}...\overline{P_n}\}$ where $\overline{P_n}$ is a partition of $\overline{C_t}$. If the partitions are entirely separate, the next state of each can be described as:

$$\overline{P_1} = g(\overline{P_1}) \tag{11.2}$$

$$\overline{P_2} = g(\overline{P_2}) \tag{11.3}$$

$$\overline{P_n} = g(\overline{P_n}) \tag{11.4}$$

However, all but the first partition take inputs from the boundaries between itself and its neighbouring partitions, so the next state of each can be described thus:

$$\overline{P_1} = g(\overline{P_1}) \tag{11.5}$$

$$\overline{P_2} = g(\overline{P_2} + \sum_{i=1}^{1} \text{Neighbouring partitions}) \tag{11.6}$$

$$\overline{P_n} = g(\overline{P_n} + \sum_{i=1}^{n-1} \text{Neighbouring partitions}) \tag{11.7}$$

As no partition takes state information from partitions it directly (or indirectly) transmits to, the feed-forward nature of the automata is preserved. Thus, provided $g()$ conforms to the requirements for the automata to converge (as laid down in previous chapters) a multiple partition automata with $g()$ as its transition function will also converge to a steady state.

11.3.2 Partition scheme

A first step in the design algorithm for irregular CA must partition the CA into groups of cells with common directions of flow of information. This is not a trivial task to automate. There are three variables to determine for each partition:

1. From which cell should the partition start disseminating state information (for a square this would be one of the corners)?

2. In which direction (NE, NW, SE, SW) should state information flow from this root cell?

3. Which cells should be part of this partition?

The solution presented below is an efficient recursive algorithm, but it will not always provide an optimum solution.

1. Find the cell furthest from the centre of mass of the automata. This is the first root.

2. Apply the following recursive function to find the optimum azimuth for the flow of information from this root.

```
azimuth = [0,0]

determine_azimuth(z,y,x):
        if there is a cell to the east of (z,y,x) in this partition, azimuth[0] =
             1
        if there is a cell to the west of (z,y,x) in this partition, azimuth[0] =
             2
        if there is a cell to the north of (z,y,x) in this partition, azimuth[1]
            = 1
        if there is a cell to the south of (z,y,x) in this partition, azimuth[1]
            = 2

        if azimuth[0] == 0:
                call the function determine_azimuth with the adjacent cell on the
                     y-axis
        if azimuth[1] == 0:
                call the function determine_azimuth with the adjacent cell on the
                     x-axis
```

Listing 11.1: Azimuth search pseudo-code

3. Apply the following recursive function to explore the partition from the root along the determined azimuth to include as many cells as possible.

```
cells_in_partition = [], bounds_of_partition = []

function explorer(z,y,x,azimuth):
        add the location (z,y,x) to cells_in_partition array
        for each of the two neighbours which this cell transmits to (determined
            by azimuth)
                call the function explorer(next_cell,azimuth)
        for each of the two neighbours which this cell receives from (determined
            by azimuth)
                if already part of a partition, or equal to zero (empty space):
                        add its location to bounds_of_partition array

```

```
11
12 explorer(first_root, azimuth)
13
14 remove repeated values from both arrays
15 if an element of the bounds array is also in the partition array, remove it
16 sort the bounds array by the cells distance from the first root
17
18 the next_root to explore from is the first element of the bounds array.
```

Listing 11.2: Partition search pseudo-code

4. While there exist cells not part of a partition, find the cell furthest from the centre of mass of the automata that lies on the boundary between a solved partition and a cell that is not part of a partition. This is the next root. Now return to step 2.

The complete source code for this algorithm can be found in appendix A.4

11.3.3 Rule generation algorithm

The state value and the directions of information transmission must be determined for each cell in the design. The directions are determined by the partition it is a part of, and whether it is a neighbour of adjacent partitions. The state value assigned to each cell is tested against the rules necessary to determine its position in the automata as described in previous chapters. Each state value is also tested against its mapping to the corresponding cell output and cell state information transmission directions. The pseudo-code for this design algorithm is listed below. For a complete listing see appendix A.4.

```
1  assignments = [], azimuth_lut = [] rules = []
2
3  for each partition:
4          for each cell in the partition:
5                  determine its inputs from the partition azimuth
6
7                  if this cell is a neighbour of a root for any other partitions:
8                          add the necessary transmission directions to the cell azimuth
9
10                 state = 0
11                 while not solved:
12                         if assignments[state] != cell_output:
13                                 state += 1;
14                                 goto start of while-loop
```

```
15                      if azimuth_lut[state] != cell_transmission_directions:
16                              state += 1;
17                              goto start of while-loop
18
19                      for each neighbouring cell to which this cell transmits:
20                              generate the next-state rule
21                              if this next_state rule violates existing rules:
22                                      state += 1;
23                                      goto start of while-loop
```

Listing 11.3: Azimuth search pseudo-code

11.3.4 Assembler

The system of cells self-assembles; no outside administration is required. Thus the assembly algorithm only describes the actions of a cell, and is common to all the cells in the automata. There are two parts to this process: one governs state information reception and its subsequent analysis, the other governs cell state information transmission.

- **State reception and analysis**.

  1. The cell requests state information from all four of its immediate neighbours.
  2. One neighbour is chosen on each axis. If state information is available from two cells on the same axis, priority is assigned to the cells closest to the north-east (an arbitrary choice).
  3. This neighbour state information is used to look-up the next state of the cell from its state look-up table (also common to each cell).
  4. The present state of the cell determines (via a look-up table) the output of the cell.
  5. The present state of the cell also determines (via a look-up table) in which directions it will transmit its state information.

- **State transmission**

1. If state information is requested from a cell, it first determines in which direction the request is coming from.
2. If the request is coming from a direction along which the cell transmits information, the state of the cell replies to the request with its current state; otherwise the cell replies with a null value.

The complete source code for this algorithm can be found in appendix A.4

11.3.5 Results

Figure 11.6 shows the partitions of a 1000 cell 2D system of cells. Figure 11.7 shows this system self-assembling and converging to the desired stable shape. Figure 11.8 shows this same system self-healing from a corrupted shape.

Implementing the algorithm of chapter 6 using a python script, the derivation for this 1000 cell self-assembling system took 30 seconds on a 3.2GHz Intel Xeon processor.

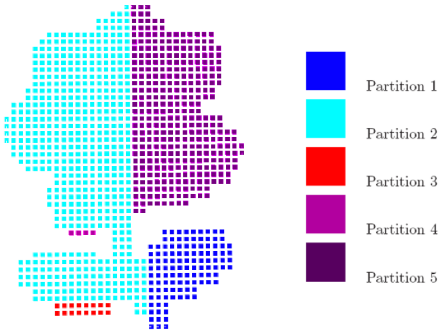


Figure 11.6: The partitions of a 2D array of 1000 cells

11.4 Design of irregular 3D automata

To adapt the design algorithm so far presented for the design of 3D systems is straightforward. To expand the analysis presented in section 3 to consider the

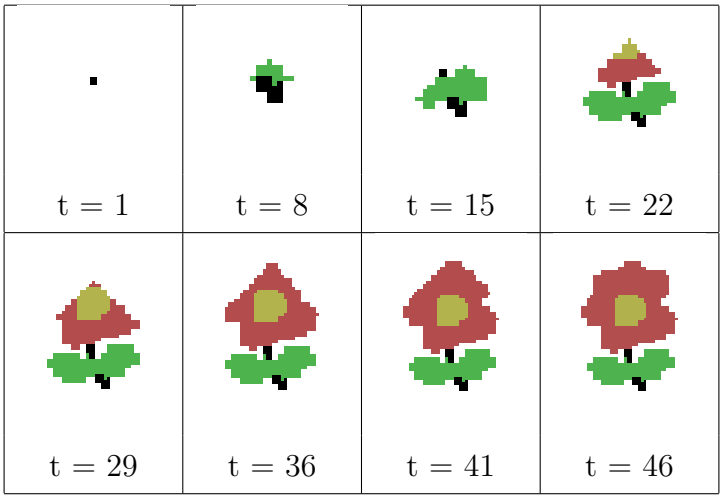


Figure 11.7: The array self-assembling and converging from the origin cell

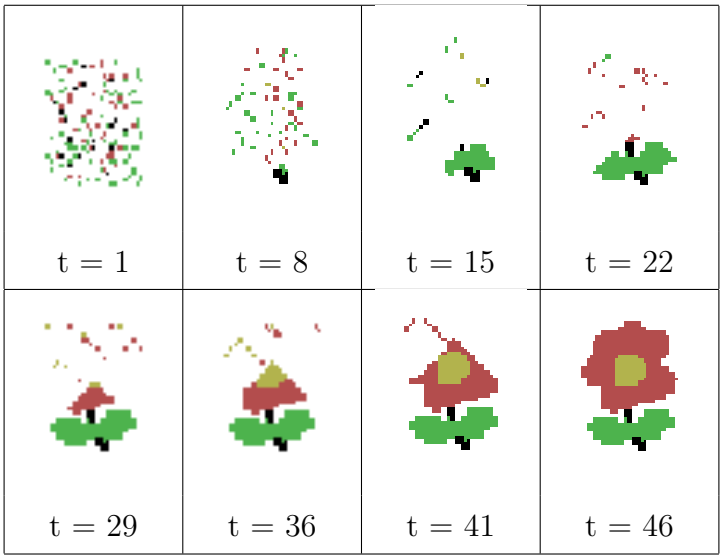


Figure 11.8: The array converging from a corrupt pattern

convergence of 3D automata we must replace the row-major vector representation of the automata with a row-column-major vector. This analysis shows that for a 3D automata to converge each cell must determine its next state according to the present state of, at most, three neighbours: one per axis.

Thus $f()$, the transition function used by each cell to determine its next state, must be a function of three variables, and the design algorithm of chapter six must be adapted to reflect this.

While adapting the design algorithm is trivial, generating the shape for it to design

is not. Whereas in two dimensions we could use readily available images, models in three dimensions are typically stored as vector diagrams — the automata needs a three-dimensional bitmap. The process for generating these bitmaps is:

1. Design, or acquire from a large freely available online stock, a three-dimensional vector model in Google's Sketchup modelling package.
2. In order to create slices along the x-axis, create a large white rectangle in the y-axis and the z-axis.
3. Take a screenshot, this image is the first slice of the bitmap of the model.
4. Move the white rectangle along the x-axis, delete any components of the model that are not immediately adjacent to the rectangle.
5. Repeat to (3) until the bitmap is complete.
6. Use a simple python script to generate a three-dimensional array of values from the slice images created. This is the three-dimensional bitmap for the design algorithm.

Figure 11.9 shows the partitions of a 3D irregular shape (Marvin the robot [KA05]). Figure 11.10 shows a system of identical cells self-assemble into this shape. Figure 11.11 shows the same system self-repair after being corrupted.

Implementing the algorithm of chapter 6 using a python script, the derivation for this 55,000 cell 3D self-assembling system took 12 hours on a 3.2GHz Intel Xeon processor.

11.5 Metamorphosis and self-assembling systems

Metamorphosis of biological systems refers to a sudden change in the form of an animal via cell growth, differentiation and apoptosis. Notable examples include a tadpole turning into a frog, and a caterpillar turning into a butterfly.

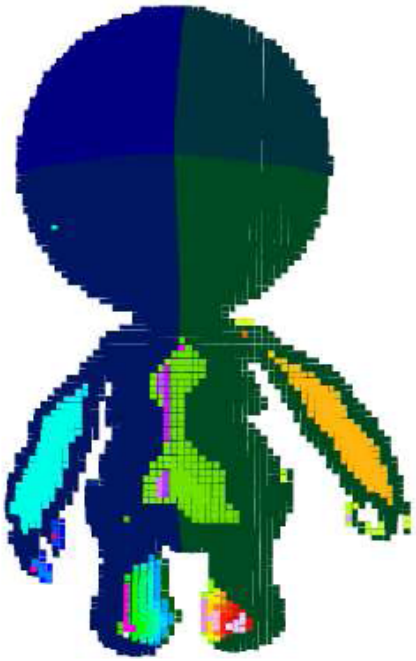


Figure 11.9: The partitions of a 3D robot shape

Metamorphosis in modular robotics refers to a self-reconfiguration. This can be seen as a minimalistic approach to designing versatile robots that can support multiple modalities of locomotion and manipulation. The following are a few potential capabilities of metamorphosis:

1. Growing to an *a priori* unknown scale, e.g. civil structures in time of emergency [PSC96].

2. Optimise gait for varying terrain: a snake-like slither for tunnels [BBR01], six legs to traverse rough terrain, long legs to climb stairs and wheels for flat surfaces [KR99].

3. Obstacle avoidance in constrained, unstructured, environments.

4. Modifying manipulars to grip objects of different shapes and sizes.

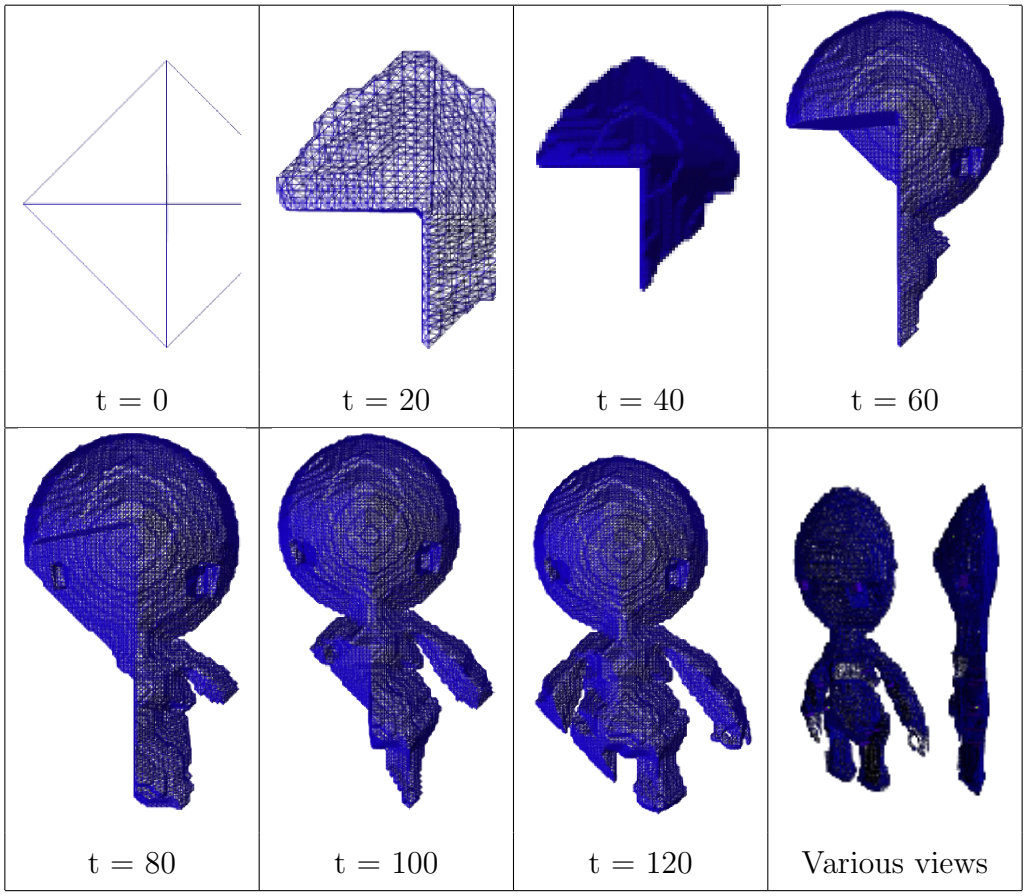


Figure 11.10: A 3D system of 55,000 cells self-assembling from the origin cell

11.5.1 Designing systems to metamorphosise

The designs presented so far have required the boundary conditions of the automata to be zero. If the bounds were set to another value, the automata would converge to a different state. Thus to design an automata to converge to one of multiple states according to its boundary conditions, the design flow would be:

1. Set the boundary conditions to correspond to the first converged state.
2. Design the automata for this state.
3. Set the boundary conditions to correspond to the next converged state.
4. Design the automata for this next state, building on existing rules and state output-azimuth assignments.

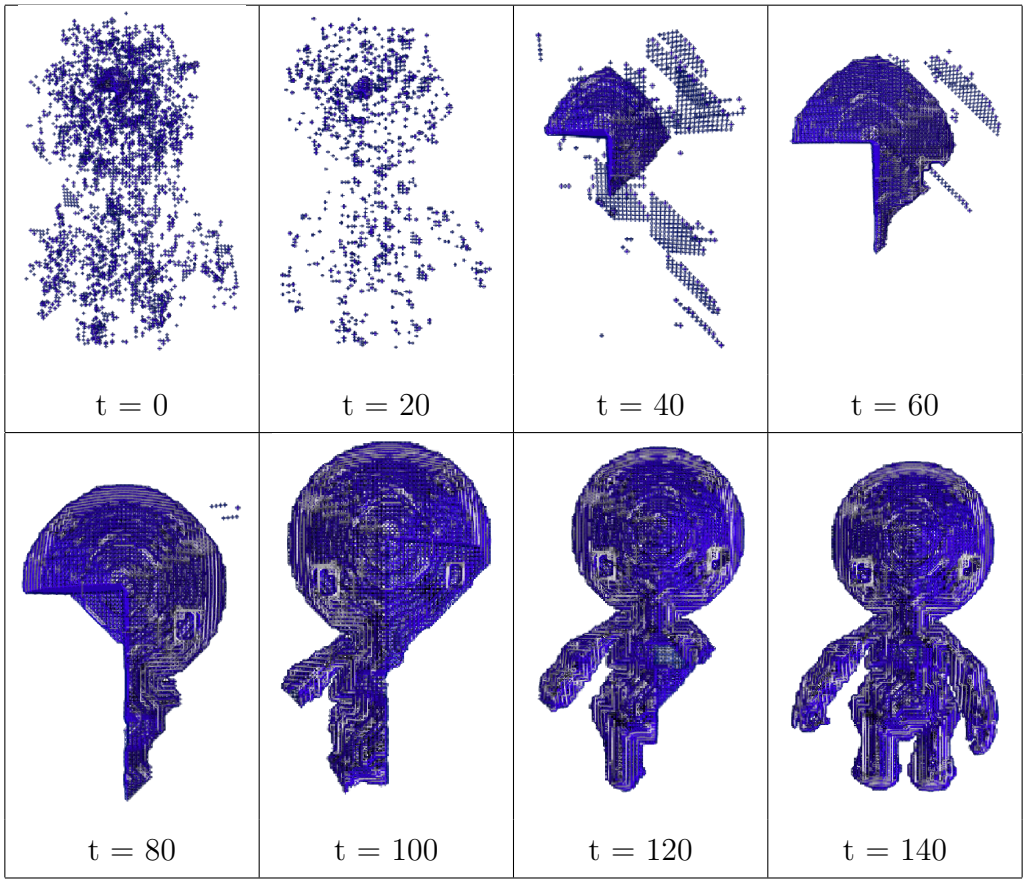


Figure 11.11: The same 3D system self-healing from a corrupt shape

5. Repeat to (3) for each of the remaining converged states.

Because the locations of every cell in the first design do not need to be populated in the second design, the algorithm also needs a cell death trigger. In the following analysis we will use Miller's [MB03] rule: a cell dies if all of its neighbouring cells are also dead.

11.5.2 Results

The ability to metamorphosise was characterised in the movie "Transformers". The following 3D designs are loosely inspired by the hero of this film "bumblebee", who could turn from a car into a robot and vice versa.

The designs are rendered using the Mayavi rendering engine and a script interface

(see appendix A.4 for the code).

Figure 11.12 shows various views and notable features of the car model.

Figure 11.13 shows various views and notable features of the robot model.

Figure 11.14 shows the 35 partitions of the car model.

Figure 11.15 shows the 39 partitions of the robot model.

Figure 11.16 shows the assembly process (from null initial conditions) of the robot model.

Figure 11.17 shows the robot model metamorphosising into the car model.

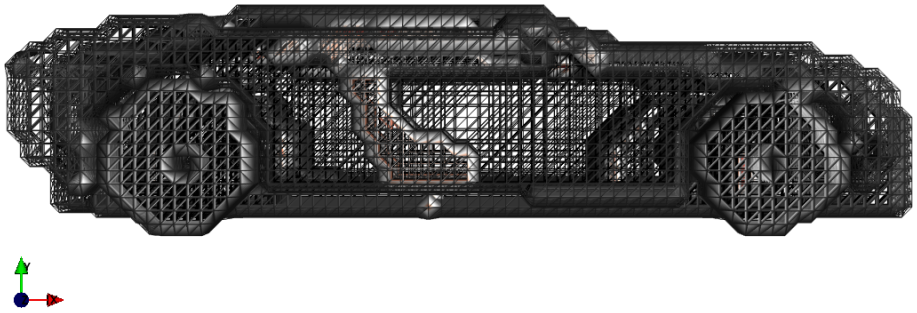
Part of the assembly algorithm is a cell-by-cell implementation of the following test: if the cell has no inputs (i.e. it is disconnected from other cells) then it should die. This can be seen in the robot-to-car example, where a significant number of the robot cells become disconnected and "die" during the transition. The car then "grows" new cells where required. If instead "death" and "growth" are metaphors for joining a pool of spare cells, this algorithm can be seen as a partial implementation of the "melt and grow" algorithm discussed previously.

In order to reduce the number of cell deaths and thus the time required to metamorphosise, we could adapt the cell death trigger by introducing a delay (e.g. "if all your neighbours are dead, wait for two clock cycles before dying too"). However, this also introduces potential feedback loops between old and new cells, creating spurious cell groupings (see figures 11.18 and 11.19).

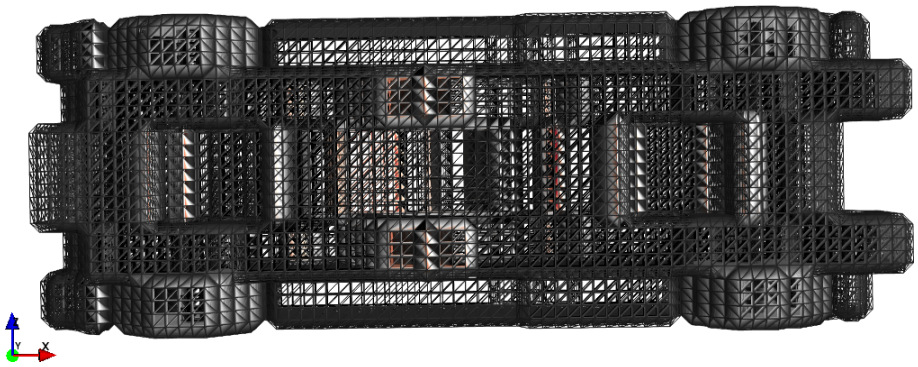
Figure 11.20 shows the self-assembly of the car model from null initial conditions.

Figure 11.21 shows the car model metamorphosising into the robot model.

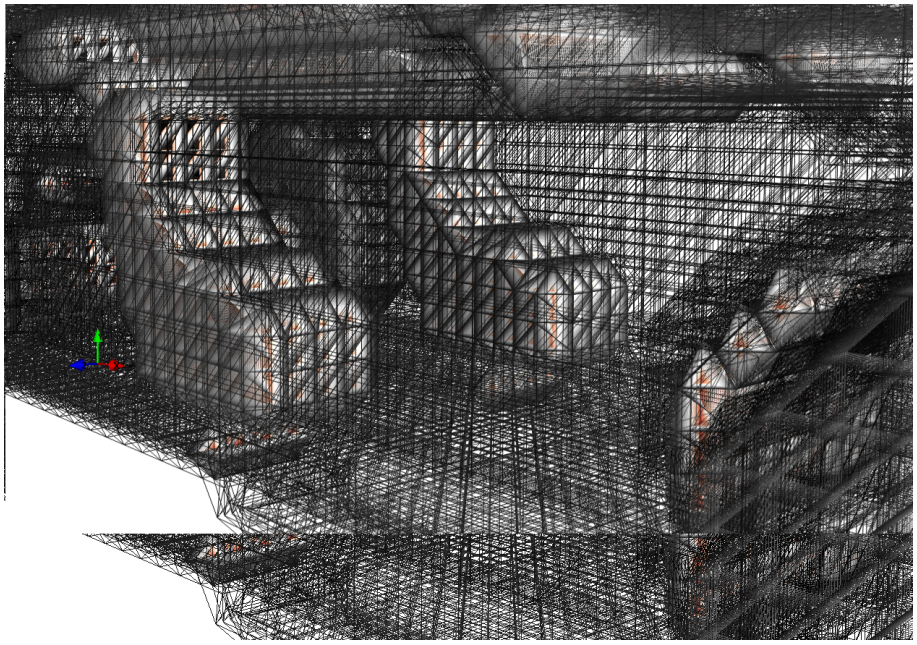
In this metamorphosis there is less evidence of cell death. This is perhaps because the car is a more cohesive design than the robot.



Side view

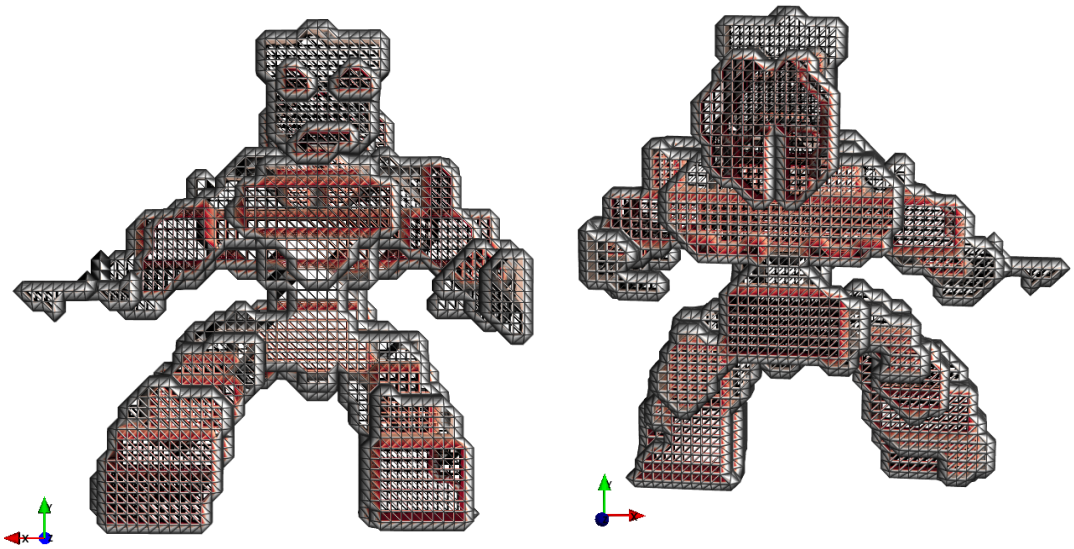


Bottom view



Interior view

Figure 11.12: Views of the “Bumblebee” car



Front view Back view

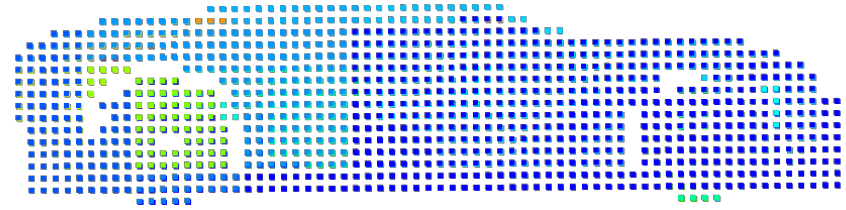
Figure 11.13: Views of the 'bumblebee' robot

In changing from the robot to the car, limbs become disconnected and begin to die at a rate that is faster than the subsequent assembly algorithm of the car. In changing from the car to the robot there are fewer extrema, so much of the cells of the car are incorporated into the robot assembly process before they have a chance to die.

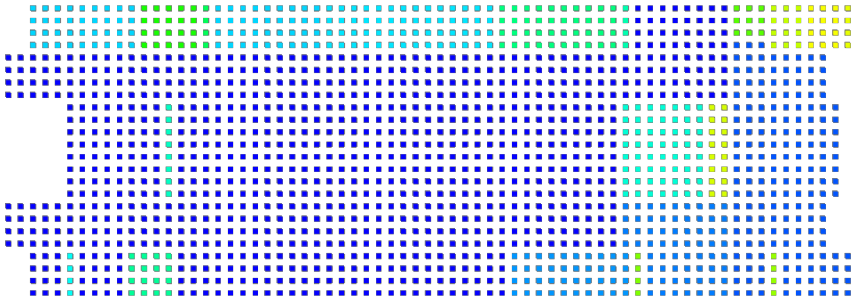
11.6 Conclusions

Attempts to engineer biological chemical and nano systems to self-assemble to a particular form have been limited by the difficulties of creating specific modules. If overcome, the limited complexity of each module will restrict any attempts to implement complicated self-assembly algorithms. Thus the bio-inspired minimalist strategy presented in this chapter may be a more appropriate scheme.

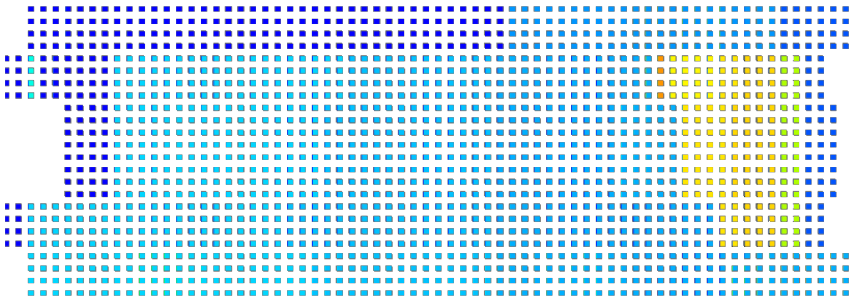
Attempts to engineer electromechanical systems to self-assemble are currently limited by the difficulty of manufacturing the complicated locomotive and inter-module bonding mechanisms. However, the study of micro-electromechanical systems (MEMS)



Side view of car partitions

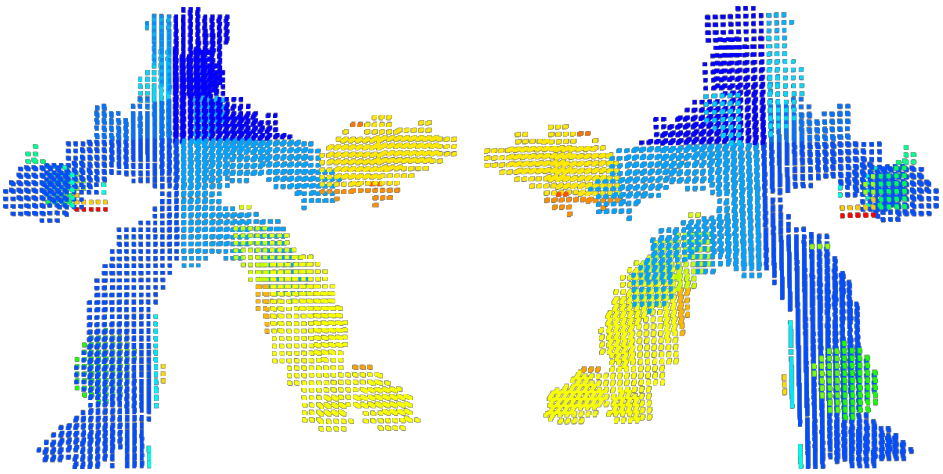


Bottom view of car partitions



Top view of car partitions

Figure 11.14: Partitions of the car model



Front partitions Back partitions

Figure 11.15: Partitions of the robot model

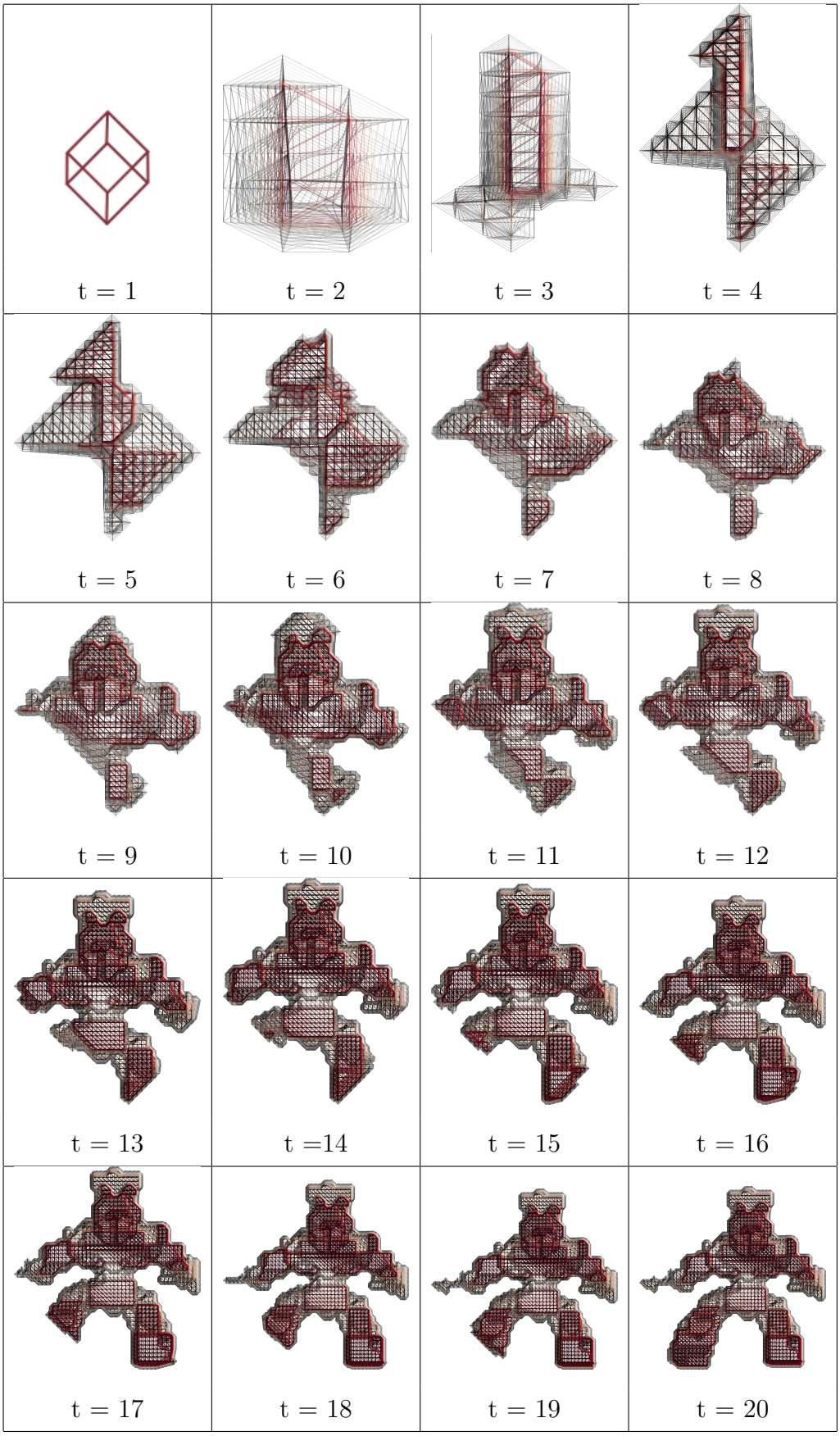


Figure 11.16: The self-assembly of a 12,000 cell robot

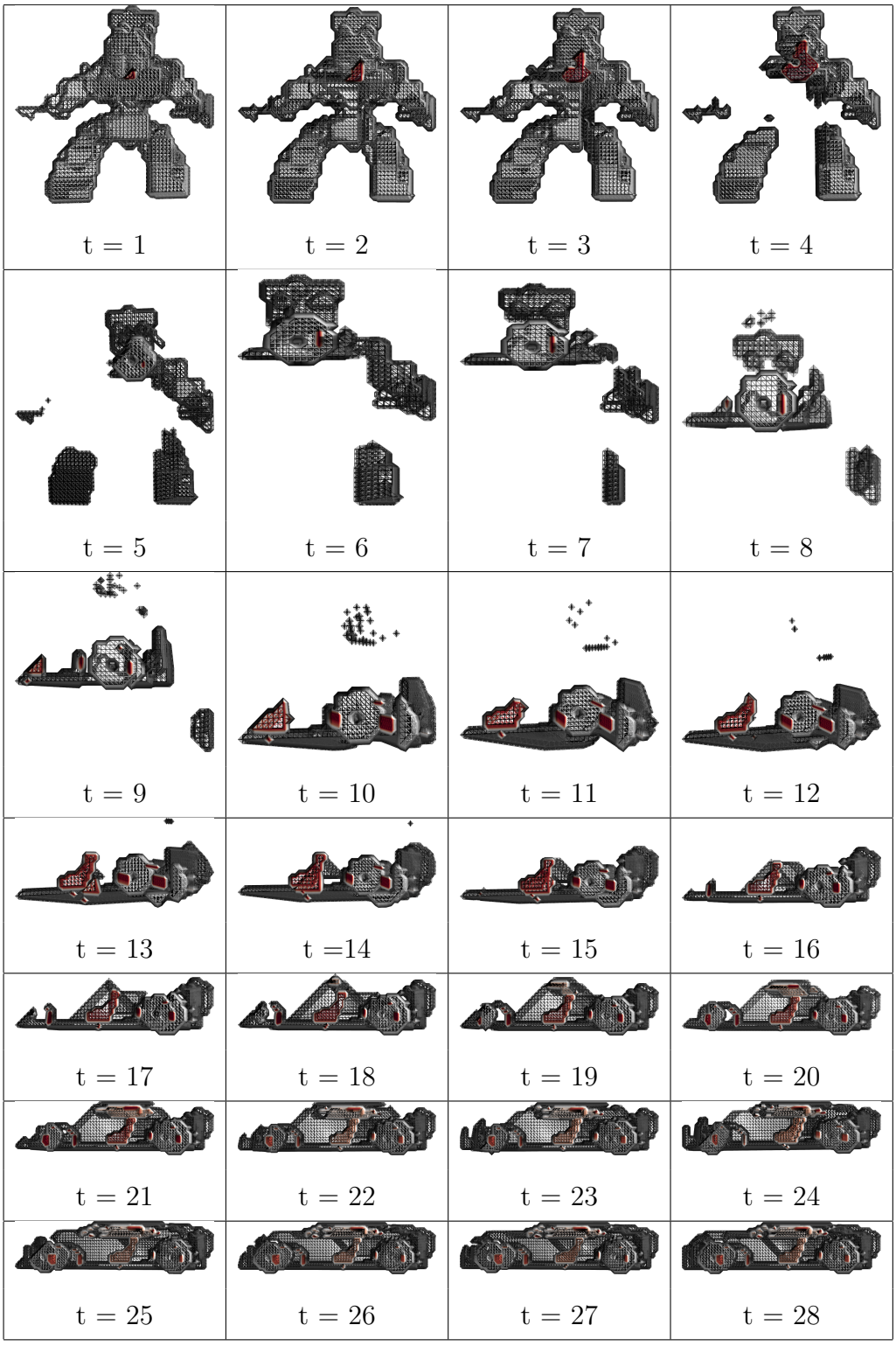


Figure 11.17: The metamorphosis of a 12,000 cell robot into a car

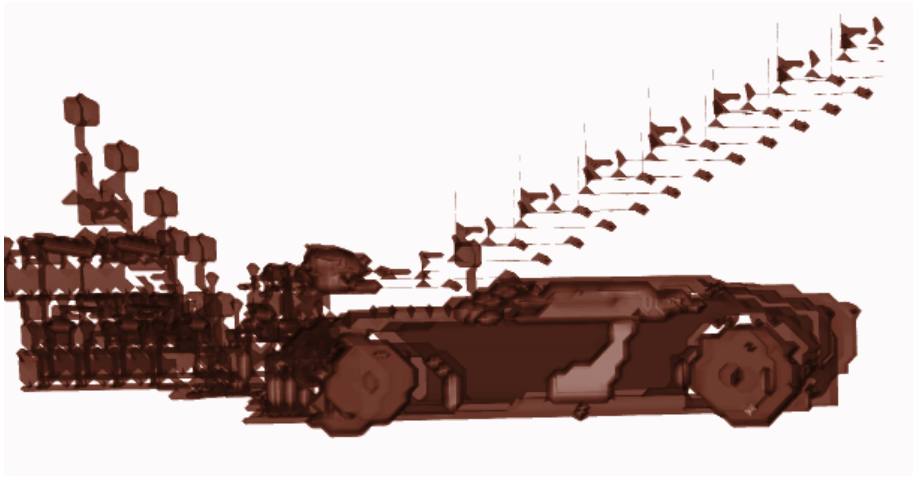


Figure 11.18: A car that has been corrupted from the robot-car transition

is making progress in the development of small, easy to manufacture components capable of locomotion and gripping. As smaller components become possible, the emphasis of the self-assembly algorithms will be on simpler schemes capable of assembling large numbers of components. Thus, the ability to co-ordinate the self-assembly of thousands of cells into a cohesive structure may be applicable.

The metamorphosis capabilities demonstrated here are at present without a practical application. However, the trend towards large distributed systems has to-date resulted in cumbersome networks of modules that are slow to adapt to changing environments. Without some adaptive abilities, self-assembling robots with thousands of components will be less effective in changing environments.

Evidence of excessive cell death and growth in the metamorphosis results suggests there are improvements to be made in the design process. Possibilities include: considering the relative location of the root cell of each design, granting each cell a slower death (i.e. waiting a few cycles before starting apoptosis), and converging to an intermediate state with fewer extrema.

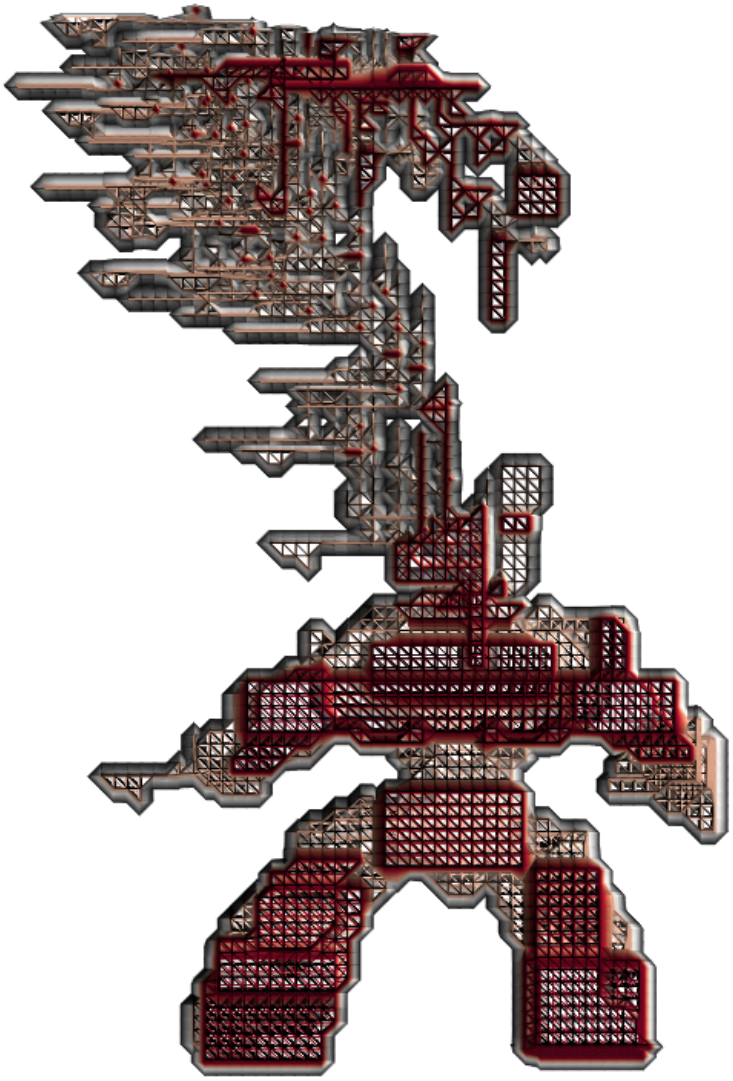


Figure 11.19: A robot that has been corrupted from the car-robot transition

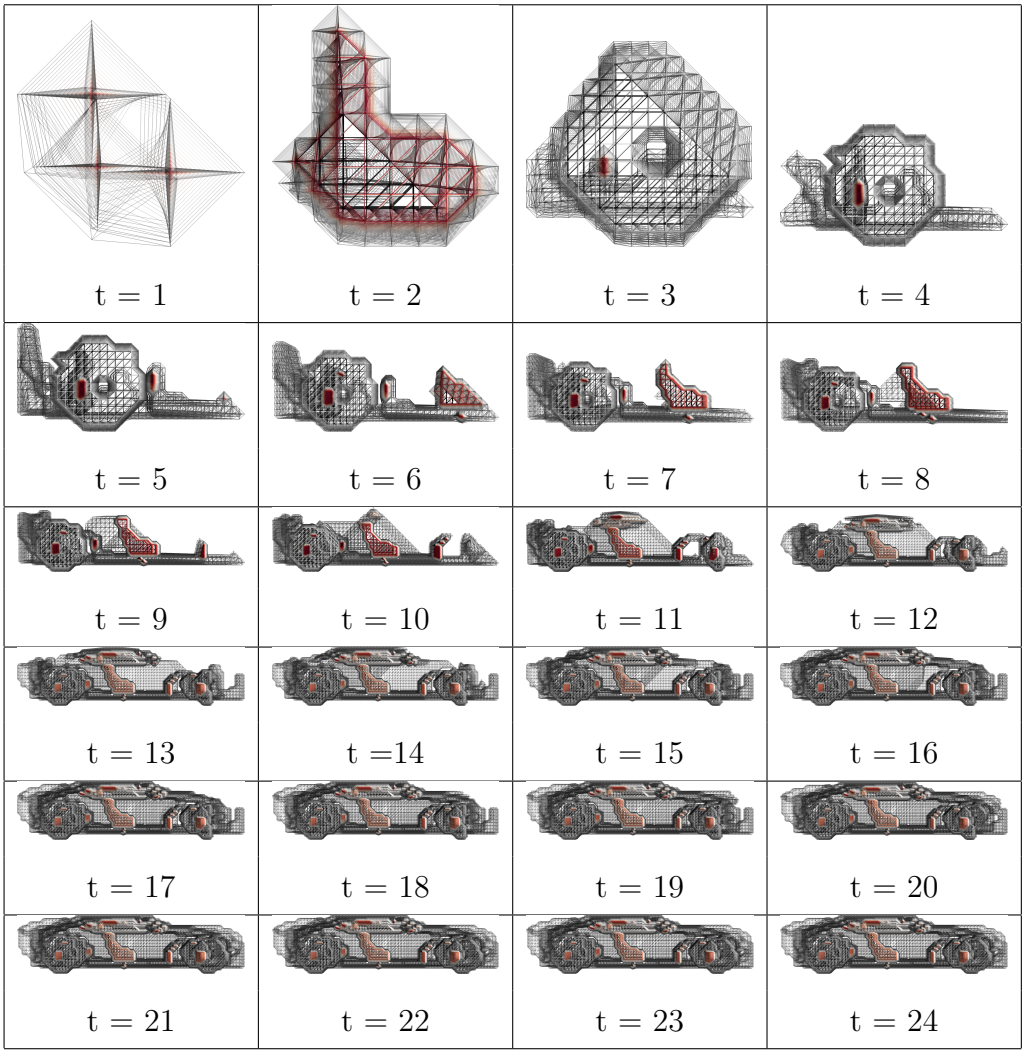


Figure 11.20: The self-assembly of a 12,000 cell car

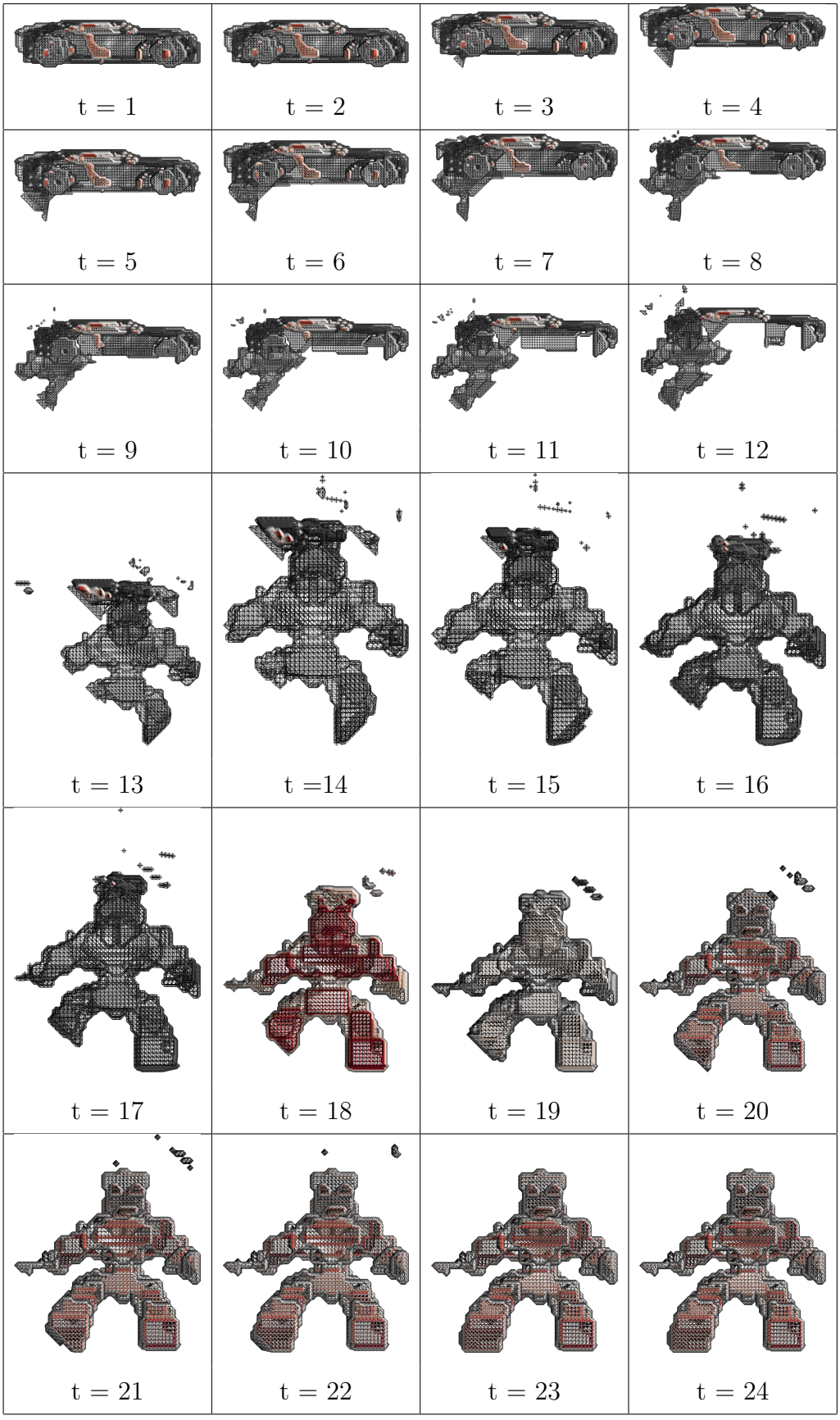


Figure 11.21: The metamorphosis of a 12,000 cell car into a robot

Chapter 12

Conclusion

We began with a brief discussion, model and analysis of an island populated by cannibals and missionaries. Simulations showed that the difference between the mobility and reproduction habits of the two populations created a clumping pattern of cannibals surrounded by missionaries. Turing [Tur50a] used this model to introduce morphogenesis as an explanation for the co-ordination of the differentiation of cells in developing and repairing biological systems.

The objective of the research described in this thesis was to imitate morphogenesis *in silico*. This with the goal of creating self-assembling and self-repairing electronic devices. Previous attempts to mimic morphogenesis have had some success.

Fleischer and Barr [FB93] imitated morphogenesis on an electronic "developing" system and demonstrated that some configurations could tolerate and repair a little corruption. Miller [Mil00] and Liu [LMT04] simplified the model of Fleischer and Barr and, using an evolutionary design algorithm, designed systems that could tolerate and repair up to 25% corruption.

We sought to incorporate an intermediate step in the mapping from biology to electronics, namely that of cellular automata. The results of this analysis are six-fold:

1. That to ensure the automata converges to the same state, regardless of initial conditions, it is necessary that the next state rule of each cell is independent of the current cell state. In addition, the next state rule can only depend upon the state of one cell per axis, either the cell to the left or to the right, either the cell above or the cell below. Thus a 1D system has one input to the next state rule, a 2D system has two inputs.

2. That it is possible to find the mapping from local rules to global arrangements using sets of linear simultaneous equations or a deterministic design algorithm.

3. That introducing redundant cell states and a state-output mapping (many-to-one) makes more convergent automata states possible. Worst-case automata-states require sufficient redundancy to uniquely identify each cell of the system.

4. That by making the automata a heterogeneous arrangement of cells of different sizes the design of self-assembling flags requires fewer redundant cell state assignations.

5. That the output of this automata can be mapped to different component types, making the design of a self-assembling self-repairing arithmetic logic unit possible. The subsequent analysis of the reliablity of this system suggests it may be more reliable than equivalent n-modular redundant systems, though further studies are required to confirm this.

6. That the cellular automata model can be adapted to design irregular shaped systems. This was demonstrated with the design of irregular robot and car shapes in three dimensions. Further, the design algorithm was shown capable of designing systems capable of metamorphosising to specific shapes depending on the boundary conditions of the automata.

Morphogenesis has been demonstrated applicable to the design of self-assembling and ultra-reliable electronic systems. Thus remains the question, where else is it applicable? Failure rates of electronic systems increase exponentially with temperature (see equation (9.7)) so perhaps morphogenesis-inspired reliability could enable

computer processors to run without needing cooling fans. Perhaps also it could be used to self-organise the behaviour of discretised computing networks, be they super-computers or smart dust. This algorithm has been applied on systems for whom longevity is essential, but it is equally applicable to the emerging field of plastic electronics. These are electronics that no longer reside on the standard FR4 composite PCBs, instead their components are laid on flexible substrates that are prone to stretching and ripping. One of the few remaining hurdles to the commercialisation of this technology is reliability - billions of plastic RFID tags cannot be printed if ten percent will fail, nor can large flexible LCD screens be rolled up if the systems cannot withstand the stretching of the substrate. Morphogenesis-inspired reliability engineering may be one means of overcoming this hurdle.

Bibliography

[AR80] J.E. Arsenault and J.A. Roberts. *Reliability and Maintainability of Electronic Systems*. Computer Science Press, Potomac, 1980.

[BBR01] Z. Butler, S. Byrnes, and D. Rus. Distributed motion planning for modular robots with unit-compressible modules. *Proceedings of IEEE International Conferance on Intelligent Robots and Systems*, pages 790–796, 2001.

[BC36] N. J. Berrill and A. Cohen. Regeneration in clavellina lepadiformis. *Journal of experimental biology*, 13, 1936.

[Ben98] J. M. Benyus. *Biomimicry: Innovation inspired by nature*. Perennial, 1998.

[BFKN98] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. 1998.

[Blo07] J. Blowey. Lecture notes for course: Mathematical biosciences. 2007.

[BOT00] D. Bradley, C. Ortega, and A. M. Tyrrell. Embryonics and immunotronics: A bio-inspired approach to fault tolerance. *2nd NASA/DoD workshop on evolvable hardware*, pages 215–2=24, 2000.

[Bow92] J. Bowles. A survey of reliability prediction procedures for microelectronics devices. *IEEE Transactions on Reliability*, 41(1):2–12, 1992.

[BR90] R. Bell and D. Reinert. Risk and system integrity concepts for safety-related control systems. *Safety-Critical Systems, Techniques and Standards*, pages 16–42, 1990.

[BS02] H. G. Beyer and H. P. Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1, 2002.

[Chi05] S. Childress. Case study 2: Turing's model of chemical morphogenesis. 2005.

[Cho01] M. Chown. *The magic furnace*. Oxford University Press, 2001.

[CL99] G. Cassandras and S. Lafortune. *Introduction to Discrete Events Systems*. Kluwer Academic Publisher, 1999.

[CMST00] O. S. Cesar, D. Mange, S. Smith, and A. Tyrrell. Embryonics: A bio-inspired cellular architecture with fault-tolerant properties. *Genetic Programming and Evolvable Machines*, 2000.

[Con05] Connect. Mobile phone reliability. *Connect*, 6, 2005.

[Cut56] F. Cutry. *The flight of birds*, volume Leonardo da Vinci. Reynal and Company, 1956.

[DDD08] M. Dean, R. D'Andrea, and M. Donovan. A self-assembling chair. *www.raffaelo.name*, 2008.

[DM91] L. D. Davis and M. Mitchell. Handbook of genetic algorithms. 1991.

[DOD90] US DOD. *MIL-HDBK-217*. 1990.

[Eco04] M. Economou. The merits and limitations of reliability predictions. *Reliability, Availability Maintainability Symposium, RAMS*, pages 352–357, 2004.

[Egg97] P. Eggenberger. Evolving morphologies of simulated 3D organisms based on differential gene expression. *Proceedings of the fourth european conference on artificial life*, pages 205–213, 1997.

[FB93] K. Fleischer and A. H. Barr. A simulation testbed for the study of multicellular development: multiple mechanisms of morphogenesis. *Artificial life III*, 1993.

[Fil05] R. Filippini. Dependability analysis of a safety critical system; the LHC beam dumping system at CERN. *CERN archive*, 2005.

[FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. Artificial intelligence through simulated evolution. 1966.

[FP90] S. Fraser and D. Perkel. Competitive and positional cues in the patterning of nerve connections. *Journal of Theoretical Neurology*, 21:51–72, 1990.

[Gal93] D. Gale. The industrious ant. *Mathematics Intelligencer*, 15(2), 1993.

[Gal98] D. Gale. *Tracking the automatic ANT and other mathematical explorations*. Springer, New York, 1998.

[Gar70] M. Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[GGJ05] S. Griffith, D. Goldwater, and J. M. Jacobson. Self-replication from random parts. *Nature*, 437, 2005.

[GL95] H. A. Gutowitz and C. Langton. Mean field theory of the edge of chaos. *Proceedings of ECAL3*, pages 52–64, 1995.

[Gri76] A. J. F. Griffiths. *An introduction to genetic analysis*. W. H. Freeman and Company, 1976.

[Gur68] J. B. Gurdon. Changes in somatic cell nuclei inserted into growing and maturing amphibian oocytes. *Journal of Embryology and Experimental Morphology*, 20:401–14, 1968.

[GVK87] H. A. Gutowitz, J. D. Victor, and B. W. Knight. Local structure theory for cellular automata. *Physica D*, pages 18–48, 1987.

[Gwe93] L. Gwennap. Estimating IC manufacturing costs. *Proceedings of the 31st annual conference on Design automation*, Microprocessor Report:15, 1993.

[HG90] N. D. Hopwood and J. B. Gurdon. Activation of muscle genes without myogenesis by ectopic expression of myod in frog embryo cells. *Nature*, pages 197–200, 1990.

[Hoh68] F. Hohn. *Applied automata theory.* Electrical science. Academic press, 1968.

[IBN96] U. Isaksen, J.P. Bowen, and N. Nissanke. *System and Software Safety in Critical Systems.* 1996.

[JA01] P. Jantapremjit and D. Austin. Design of a modular self-reconfigurable robot. *Proceedings of Australian conference on robotics and automataion*, 2001.

[KA05] K. Kirkpatrick and D. Adams. The hitchhiker's guide to the galaxy. *Touchstone Pictures*, 2005.

[Kel95] Kevin Kelly. *Out of control: the new biology of machines, social systema and the economic world.* Basic Books, 1995.

[KGPV08] N. Krasnogor, S. Gustafson, D.A. Pelta, and J.L. Verdegay. *Systems self-assembly: multidisciplinary snapshots*, volume 5. 2008.

[KR99] K. Kotay and D. Rus. Locomotion versatility through self-reconfiguration. *Robotics and Autonomous Systems*, 26:217232, 1999.

[KRVM98] K. Kotay, D. R., M. Vona, and C. McGray. The self-reconfiguring robotic molecule. *Proceedings of IEEE International Conference on Robotics and Automation*, 1998.

[Lan86] C. Langton. Studying artificial life with cellular automata. *Physica D.*, 22:120–149, 1986.

[Lan90] C. Langton. Computation at the edge of chaos. *Physica D*, 42, 1990.

[Lev00] D. Levi. Hereboy: a fast evolutionary algorithm. *Proceedings of the 2nd NASA/DoD workshop on evolvable hardware*, pages 17–24, 2000.

[LKY+96] X. Liu, C. Kim, J. Yang, R. Jemmerson, and X. Wang. Induction of apoptotic program in cell-free extracts: requirement for datp and cytochrome c. *Cell*, 86(1):147–57, 1996.

[LMT04] H. Lui, J. F. Miller, and A. Tyrrell. An intrinsic robust transient fault-tolerant developmental model for digital systems. *Proceedings of GECCO*, 2004.

[MB03] J. F. Miller and W. Banzhaf. Evolving the program for a cell: From french flags to boolean circuits. *On Growth, Form and Computers*, 2003.

[Mei82] H. Meinhardt. *Models of biological pattern formation.* Academic Press, London, 1982.

[Mil00] J. Miller. Principles in the evolutionary design of digital circuits, part 1. *Journal of genetic programming and evolvable machines*, 1, 2000.

[MKTK99] S. Murata, H. Kurokawa, K. Tomita, and S. Kokaji. Self-assembly and self-repair method for a distributed mechanical system. *IEEE Transactions on Robotics and Automation*, 1999.

[MMSD07] A. Moglia, A. Menciassi, M. O. Schurr, and P. Dario. Wireless capsule endoscopy: from diagnostic devices to multipurpose robotic systems. *Biomedical Microdevices*, 2007.

[Mor04] T. H. Morgan. An attempt to analyse the phenomena of polarity in tubularia. *Journal of experimental Zoology*, 1:587–591, 1904.

[MSP+89] E. E. Moore, S. R. Shackford, H. L. Pachter, J. W. McAninch, and M. L. Ramseonfsky. Organ injury scaling: spleen, liver and kidney. *Journal of Trauma*, 29(12):1664–6, 1989.

[Neu66] J. Neumann. Theory of self-reproducing automata. *University of Illinois press*, 1966.

[NKC03] R. Nagpal, A. Kondacs, and C. Chang. Programming methodology for biologically-inspired self-assembling systems. *AAAI Spring Symposium on Computational Synthesis*, 2003.

[oI08] The Technical Museum of Innovation. Robotics: about the exhibition. *www.thetech.org*, 2008.

[OOAB81] G. M. Odell, G. Oster, P. Alberch, and B. Burnside. The mechanical basis of morphogenesis. *Developmental biology*, 85:446–462, 1981.

[PN94] M. G. Pecht and F.R. Nash. Predicting the reliability of electronic equipment. *Proceedings of the IEEE*, 82(7):992–1004, 1994.

[PSC96] A. Pamecha, D. Stein, and G. Chirikjian. Design and implementation of metamorphic robots. *Proceedings of the 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*, pages 1–10, 1996.

[RAC97] RAC. *Electronic Part Reliability Data*. RAC, Rome (NY) USA, 1997.

[RHT+97] A. L. Rosee, T. Hader, H. Taubert, R. Rivera-Pomar, and H. Jackle. Mechanism and bicoid-dependent control of hairy stripe 7 expression in the posterior region of the drosophila embryo. *The Embryology journal*, 16, 1997.

[RO00] J. D. Robertson and S. Orrenius. Molecular mechanisms of apoptosis induced by cytotoxic chemicals. *Critical reviews in toxicology*, 30(5):609–627, 2000.

[RV01] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10:107–124, 2001.

[Sch05] D. Schutz. Lecture notes: Analysis year 4. 2005.

[Sha06] A. Shapiro. Ultra-reliability at nasa. *44th AIAA Aerospace Sciences Meeting and Exhibit*, 2006.

[Shi97] M. Shipper. The firefly machine. *Evolution of parallel cellular machines*, 1997.

[Sho68] M. L. Shooman. *Probabilistic Reliability: an Engineering Approach*. Mc-Graw Hill, 1968.

[SWMK05] C. Schinkel, M. Wick, G. Muhr, and M. Koller. Analysis of systemic interleukin-11 after major trauma. *Shock*, 2005.

[TM90] T. Toffoli and N. Margolus. Invertible cellular automata: A review. *Physica D*, 45:229–253, 1990.

[Tou99] D. N. A. Touba. A low cost approach to detecting, locating and avoiding interconnect faults in FPGA-based reconfigurable systems. *Proceedings of the International conference VLSI design*, 1999.

[Tri82] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science applications*. Prentice-Hall, New York, 1982.

[TSAT03] Y. Thoma, E. Sanchez, J. M. Arostegui, and G. Tempesti. A dynamic routing algorithm for a bio-inspired reconfigurable circuit. *Lecture notes in computer science*, 2778:681–690, 2003.

[Tur50a] A. M. Turing. The chemical basis of morphogenesis. *Philosophical transactions of the Royal Society*, Ser. B 237, 37, 1950.

[Tur50b] A. M. Turing. Computer machinery and intelligence. *Mind*, 1950.

[Wat92] G. Watson. MIL reliability: a new approach. *IEEE Spectrum*, pages 46–49, 1992.

[Wil] D. J. Wilkins. www.weibull.com. *Weibull analysis*.

[Wol69] L. Wolpert. Positional information and the spatial pattern of cellular differentiation. *J Theor. Biol.*, 25:1-47, 1969.

[Wol82] S. Wolfram. Cellular automata as simple self-organising systems. *Caltech preprint*, 1982.

[Wol02] S. Wolfram. *A new kind of science*. Wolfram media Inc., 2002.

[Wyl80] A. H. Wyllie. Glucocorticoid-induced thymocyte apoptosis is associated with endogenous endonuclease activation. *Nature*, 284:555–556, 1980.

[WZBL05] P. J. White, V. Zykov, J. Bongard, and H. Lipson. Three-dimensional stochastic reconfiguration of modular robots. *Proceedings of the Robotics: Science and Systems Conference*, pages 161–168, 2005.

[YWPS08] M. Yim, P. White, M. Park, and J. Sastra. Modular self-reconfigurable robots. *Encyclopedia of complexity and systems science*, 2008.

Appendix A

Source code

A.1 Reaction-diffusion models

Listing A.1: Cannibals and Missionaries code

```
#!/usr/bin python

# each player is of type (0 = missionary, 1 = cannibal)
# each player can move up, down, left or right one place, or stay where it is
# if two cannibals share a cell, they create another cannibal in the same cell
# if two missionaries and a cannibal share a cell, the cannibal becomes a missioanry

# the island is a 128*128 cell square
grid_dimensions = (128,128)
iterations = 100
initial_players = 1000

import random, math

# responsible for location and movement of players
class player:
        def __init__(self,x,y,type):
                self.x = x
                self.y = y
                self.type = type
        def move_player(self):
                rand = random.random()
                if rand < 0.2:
                        self.x += 1;
                elif rand < 0.4:
                        self.x -= 1;
                elif rand < 0.6:
                        self.y += 1;
                elif rand < 0.8:
                        self.y -= 1;
                if(self.y == grid_dimensions[1]): self.y = 0
                elif(self.y < 0): self.y = grid_dimensions[1] - 1
                if(self.x == grid_dimensions[0]): self.x = 0
                elif(self.x < 0): self.x = grid_dimensions[0] - 1

                return (self.x,self.y)

# randomnly place 'initial_players' number of players across grid
players = []
def initializePlayers():
        print "starting initialization"
        for i in range(initial_players):
                type = random.random()
                if type > 0.5: type = 1
                else: type = 0
                x = math.floor(random.random()*grid_dimensions[0])
                y = math.floor(random.random()*grid_dimensions[1])
                players.append(player(x,y,type))

# detect collisions
cperson = (0,0)
def position_match(person):
        if cperson[0] == person.x and cperson[1] == person.y:
                return 1;
        else:
                return 0;

# update cell with player in it
def check_cell(person,players):
        cperson = (person.x,person.y)
        sharingCell = filter(position_match,players)
        if len(sharingCell) > 1:
                cannibals = 0
                missionaries = 0
                for person in sharingCell:
                        if person.type == 0: missionaries += 1
                        elif person.type == 1: cannibals += 1
                        if cannibals == 2:
                                cannibals = 0
                                players.append(player(person.x,person.y,1))
                        if missionaries == 2 and cannibals == 1:
                                missionaries = 0
                                cannibals = 0
                                for i in range(len(sharingCell)):
```

```
                                        if sharingCell[i].type == 1:
                                                sharingCell[i].type = 0


def type1(person):
        return person.type

import Image
def main():

        # initialize players
        initializePlayers()

        # iterations of simulation start here
        for i in range(iterations):
                print "no of cannibals ",len(filter(type1,players))
                print "total players ",len(players)
                cannibals_to_add = 0
                print "starting iteration ",i

                # move all the players
                for person in players:
                        person.move_player()

                # create new image of grid
                image = Image.new("RGB",(grid_dimensions[0],grid_dimensions[1]))
                pix = image.load()

                # for each cell in grid
                for y in range(grid_dimensions[1]):
                        for x in range(grid_dimensions[0]):
                                pix[x,y] = (0,0,0)
                                missionaries = 0
                                total = 0
                                cannibals_to_add = []
                                cannibals_to_convert = 0

                                # for each player in grid
                                for person in players:
                                        if person.x == x and person.y == y:
                                                total += person.type

                                                # create missionaries
                                                if person.type == 0: missionaries += 1;
                                                if total == 2:
                                                        total = 0
                                                        cannibals_to_add.append((x,y))
                                                if missionaries == 2 and total == 1:
                                                        cannibals_to_convert += 1
                                                        missionaries = 0
                                                        total = 0
                                                        pIndex = 0

                                                # convert cannibals
                                                while(cannibals_to_convert > 0):
                                                        if players[pIndex].type
                                                            == 1:
                                                                cannibals_to_convert
                                                                    = 0
                                                                players[pIndex].
                                                                    type = 0
                                                        else:
                                                                pIndex += 1
                                for cannibal in cannibals_to_add:
                                        players.append(player(cannibal[0],cannibal[1],1))

                # pixelate image
                for person in players:
                        if person.type == 0: pix[person.x,person.y] = (0,255,0)
                        if person.type == 1: pix[person.x,person.y] = (255,0,0)

                image.save("output"+str(i)+".png","PNG")
main()
```

Listing A.2: Reaction-diffusion example code

```
#!/usr/bin python

# each cell has two chemical concentrations, a, b
# each cell computes its CHANGE in concentration with the formulae:

# a = a + s(16-ab)+D(SUM(a of neighbours) - 4a)
# b = b + s(ab-b-B)+D(SUM(b of neighbours) - 4b)
# B = Bc +- BcVar

# initial conditions
ae = 4.0
be = 4.0

# diffusion constants
Da = 0.25
Db = 0.0625
s = 0.03125
s = 0.04525
Bc = 12.0
BcVar = 0.2

# simulation settings
grid_dimensions = (64,64)
iterations = 2000

import random, math, Image
def main():
        global s

        # initialise simulation
        initializeCells()

        # simulation iterations start here
        for i in range(iterations):

                # diffuse chemicals
                updateCells()
                min = 10
                max = 0
                for k in cells:
                        if(k.b<min): min=k.b
```

```
                        if(k.b>max): max=k.b

                # create image
                outputImage(cells,i,max,min)
                print i,min,max

# responsible for chemical concentrations of a cell
class cell:
        def __init__(self,x,y,B):
                self.a = ae
                self.b = be
                self.x = x
                self.y = y
                self.B = B

        # determine next concentrations of each cell
        def calculateConcentrations(self,a,b):
                self.a = self.a - s*(16.0-(self.a*self.b))+Da*(reduce(lambda x, y: x+y, a
                    ) - 4.0*self.a)
                self.b = self.b - s*(-1*self.a*self.b+self.b-self.B)+Db*(reduce(lambda x,
                     y: x+y, b) - 4.0*self.b)

cells = []

# randomnly initialize concentrations at each cell
def initializeCells():
        del cells[:]
        for y in range(grid_dimensions[1]):
                for x in range(grid_dimensions[0]):
                        B = Bc + ((random.random()-0.5)*2*BcVar)
                        cells.append(cell(x,y,B))

# for each cell, determine its neighbours, then update its chemical concentrations
def updateCells():
        for i in range(len(cells)):
                if(i>(grid_dimensions[0]-1)): index_up = i-grid_dimensions[0]
                else: index_up = i+(grid_dimensions[0]*(grid_dimensions[1]-1))
                if(i<(grid_dimensions[0]*(grid_dimensions[1]-1))): index_down = i+
                    grid_dimensions[0]
                else: index_down = i % grid_dimensions[0]
                if(i % grid_dimensions[0]) != 0: index_left = i-1
                else: index_left = i+(grid_dimensions[0]-1)
                if((i-(grid_dimensions[0]-1)) % grid_dimensions[0]) != 0: index_right = i
                    +1
                else: index_right = i-(grid_dimensions[0]-1)
                if(i==0): index_right = 1;

                a=[cells[index_up].a,cells[index_down].a,cells[index_left].a,cells[
                    index_right].a]
                b=[cells[index_up].b,cells[index_down].b,cells[index_left].b,cells[
                    index_right].b]
                cells[i].calculateConcentrations(a,b)


def sumOut(x,y):
        return x.b+y.b

def maxOut(x,y):
        if x.b > y.b: return x
        else: return y
```

```
# save iteration result as an image
def outputImage(data,i,max,min):
        image = Image.new("RGB",(grid_dimensions[0],grid_dimensions[1]))
        pix = image.load()
        scalar = math.floor(255/(max-min))
        for cc in data:
                gg = ((cc.b-min)*scalar)
                pix[cc.x,cc.y] = (gg,gg,gg)
        image.save("output"+str(i)+".png","PNG")


if __name__ == "__main__":

        main()
```

Listing A.3: Reaction-diffusion Meinhardt example code

```
#!/usr/bin python

from numpy import *
import math,Image
# grid_dimensions = (y,x)
grid_dimensions = (64,64)

# diffusion coefficients
C = 0.1
A = 0.2
Dg = 0.1
Ds = 0.06
Pg = 0.3
Ps = 0.4
GA = 0.5
B = 0.6

C = 0.3
A = 0.1
Dg = 0.1
Ds = 0.06
Pg = 0.001
Ps = 0.001
GA = 0.0004
B = 0.1

# simulation parameters
iterations = 100

# randomnly initialize grid
def createArray(initial_value):
        arr = random.random(grid_dimensions)*0.3+1
        return arr

# differentiate 2d data (using python-numerical function diff)
def differentiate(data):
        aug_data = hstack((data,data))
        aug_data = vstack((aug_data,aug_data))
        return diff(aug_data,n=2)
```

```
def ddX(data,x,y):
        if x==0: x = grid_dimensions[1]
        return data[y,x+1]

# update chemical g1 concentrations
def updateG1(x,y):
        global g1,g2,r,s1,s2
        ddg1_arr = differentiate(g1)
        g1[y,x] = g1[y,x] + ((C*s2[y,x]*pow(g1[y,x],2))/r[y,x])-A*g1[y,x]+Dg*ddX(ddg1_arr
             ,x,y)+Pg

# update chemical g2 concentrations
def updateG2(x,y):
        global g1,g2,r,s1,s2
        ddg2_arr = differentiate(g2)
        g2[y,x] = g2[y,x] + ((C*s1[y,x]*pow(g2[y,x],2))/r[y,x])-A*g2[y,x]+Dg*ddX(ddg2_arr
             ,x,y)+Pg

# update chemical R concentrations
def updateR(x,y):
        global g1,g2,r,s1,s2
        r[y,x] = r[y,x] + C*s2[y,x]*pow(g1[y,x],2)+C*s1[y,x]*pow(g2[y,x],2)-B*r[y,x]

# update chemical s1 concentrations
def updateS1(x,y):
        global g1,g2,r,s1,s2
        dds1_arr = differentiate(s1)
        s1[y,x] = s1[y,x] + GA*(g1[y,x]-s1[y,x])+Ds*ddX(dds1_arr,x,y)+Ps

# update chemical s2 concentrations
def updateS2(x,y):
        global g1,g2,r,s1,s2
        dds2_arr = differentiate(s2)
        s2[y,x] = s2[y,x] + GA*(g2[y,x]-s2[y,x])+Ds*ddX(dds2_arr,x,y)+Ps

# save results of iteration as an image
def outputImage(g1,g2,s1,s2,r,i):
        image = Image.new("RGB",(grid_dimensions[1],grid_dimensions[0]))
        pix = image.load()
        if g1.max() == g1.min(): min1 = g1.max()-1
        else: min1 = g1.min()
        scalar1 = math.floor(255/(g1.max()-min1))
        if g2.max() == g2.min(): min2 = g2.max()-1
        else: min2 = g2.min()
        scalar2 = math.floor(255/(g2.max()-min2))

        for y in range(grid_dimensions[0]):
                for x in range(grid_dimensions[1]):
                        gg = ((g1[y,x]-min1)*scalar1)
                        rr = (((g2[y,x]-min2)*scalar2)+gg)/2
                        pix[x,y] = (rr,rr,rr)
        image.save("output"+str(i)+".png","PNG")



def main():
        set_printoptions(threshold=nan)

        # create arrays for each of the 5 chemicals
        global g1,g2,r,s1,s2
        g1 = createArray(1)
        g2 = createArray(1)
        r = createArray(1)
        s1 = createArray(0.1)
        s2 = createArray(0.01)

        # create initial variations at centre of grid
        g1[32,32] = 5
        g1[31,32] = 5
        g1[30,32] = 5
        g1[31,31] = 5
        g1[31,31] = 5
        g1[31,31] = 5
        g2[32,29] = 5
        g2[31,29] = 5
        g2[30,29] = 5
        g2[31,29] = 5
        g2[31,29] = 5
        g2[31,29] = 5

        # simulation iterations start here
        for i in range(iterations):
                print g1.max(), g1.min()
                print i

                # for each cell in grid, update concentrations of each chemical
                for y in range(grid_dimensions[0]):
                        for x in range(grid_dimensions[1]):
                                updateG1(x,y)
                                updateG2(x,y)
                                updateR(x,y)
                                updateS1(x,y)
                                updateS2(x,y)

                # save result as an image
                outputImage(g1,g2,s1,s2,r,i)


if __name__ == "__main__":
        main()
```

A.2 Design and test of convergent CA

Listing A.4: Python script to design and test convergent CA

```
#!/usr/bin/python

import Image, ImageDraw, ImageFont, pygame
import operator
import time
from random import randint
```

```
# development cycle parameters
rotations = [1]
display_rules_interval = 0

# test cycle parameters
random_init = 1
max_range = 6990

# output image parameters
win_width = 400
win_height = 400
prefix = "cd"

# display animation of test cycle
def displayImage(filename):
        import pygame
        background = pygame.image.load(filename).convert()
        background = pygame.transform.scale(background,(win_width,win_height))
        screen.blit(background,(0,0))
        pygame.display.update()

# save CA as image
def outputImage(filename,cell_array):
        image = Image.new("RGB",(width,height))
        pix = image.load()
        for y in range(height):
                for x in range(width):
                        try:
                                pix[x,y] = cell_array[y*width+x].dval
                        except:
                                pix[x,y] = (0,0,0)
        image.save(filename,"PNG")


# cell location and assignments
class cell:
        def __init__(self,x,y,val):
                self.x = x
                self.y = y
                self.weight = (x+y)*-1
                self.val = val
                self.dval = val
                self.solved = 0

# for storing solution parameters and rules.
class solution:
        def __init__(self,rotation,rules,assignments,cost,width,height,flag):
                self.rotation = rotation
                self.rules = rules
                self.assignments = assignments
                self.cost = cost
                self.width = width
                self.height = height
                self.flag = flag

import sys,getopt
options = ''
if len(sys.argv) > 1:
    for opt,arg in getopt.getopt(sys.argv[1:],
            "g:i:o:", ["jam=","img=","demo="])[0]:
        if opt == "-j" or opt == "--jam":
            text = arg
        elif opt == "-i" or opt == "--img":
            text = arg
        elif opt == "-d" or opt == "--demo":
            options = arg
else:
    text = raw_input("Enter location of existing ruleset (.jam) file or image from which
        to generate rules\r\n");

cells = []
rcells = []
flag = []
solutions = []

# load existing solution (stored as a .jam file)
if text.count('.jam') > 0:
    """Load JAM file"""
    import pickle
    print "Loading rules from ",text
    f = open(text,"r")
    solutions.append(pickle.load(f))
    width = solutions[0].width
    height= solutions[0].height
    rotation = solutions[0].rotation
    rules = solutions[0].rules
    assignments = solutions[0].assignments
    flag = solutions[0].flag
    max_range = len(assignments)
    print "Cell matrix dimensions ",width,height
    print "Assembling using rotation ",rotation
    print "Solution uses ",len(assignments),"assignments and",len(rules),"rules"

# generate solution from image
else:
    # load image
    print "Loading image",text
    import Image, operator
    im = Image.open(text)
    print im.format, im.size, im.mode
    (width,height) = im.size

        # load image into an array of cells
    for y in range(height):
                row = []
                for x in range(width):
                        cells.append(cell(x,y,im.getpixel((x,y))))
                        row.append(im.getpixel((x,y)))
                rcells.append(row)
                flag.append(row)
assignments = {}
rules = {}

if(1):
        # surround rcells array with boundary conditions - zero
        from numpy import *
        rcells = zeros((height+2,width+2))
```

```
# sort cells by their distance from one corner
cells.sort(key=operator.attrgetter('weight'))
cells.reverse

slice = 0
rval = 0

# function for storing next-state rule in an associative array
def makeRule(i,j,l):
        if not rules.has_key(i):
                rules[i] = {}
        rules[i][j]=l

# for each cell determine its necessary next-state rule and assignments
for cc in cells:
        while(1):
                solved = [1,2]
                rval += 1
                # test against previous assignments
                try:
                        if(assignments[rval] != cc.val):
                                continue
                except KeyError:
                        pass

                def n(y,x):
                        return rcells[cc.y+y+1,cc.x+x+1]


                # get state of neighbouring cells
                ne = n(-1,1)
                e = n(0,1)
                ss = n(1,0)
                sw = n(1,-1)

                # test against existing rules
                try:
                        if rules[ne][rval] != e:
                                continue;
                        else:
                                solved[0] = 0
                except:
                        pass
                try:
                        if rules[rval][sw] != ss:
                                continue;
                        else:
                                solved[1] = 0
                except:
                        pass

                # store next-state rules
                for s in solved:
                        if s == 1:
                                makeRule(ne,rval,e)
                        elif s == 2 and cc.x == 0:
                                makeRule(rval,sw,ss)

                # store assignments
                assignments[rval] = cc.val
                rcells[cc.y+1,cc.x+1] = rval
                rval = 0
                break


        # make rule for origin cell
        makeRule(0,0,rcells[1][1])
        print "Alphabet size necessary: ",len(assignments)
        rule_size = 0
        for a in rules:
                rule_size += len(rules[a])
        print "Rule size necessary: ",rule_size



# test cycle starts here

cells = []
from numpy import *

#initialise null array of cells
icells = zeros((height+1,width+1))

# cell class, responsible for getting inputs from neighbours, determining next-state and
    output

class dcell:
        def __init__(self,x,y,init):
                self.x = x
                self.y = y
                self.val = init
                try:
                        self.output = assignments[init]
                except:
                        self.output = 0

        def getInputs(self):
                return (int(icells[self.y,self.x+1]),int(icells[self.y+1,self.x]))

        def update(self):
                (y,x) = self.getInputs()
                try:
                        self.val = rules[y][x]
                except KeyError:
                        self.val = 0
                try:
                        self.output = assignments[self.val]
                except KeyError:
                        self.output = 0

# initialise cells either with a random state or a state = zero
init = 0
for y in range(height):
        row = []
        for x in range(width):
                if init == 0:
                        row.append(dcell(x,y,0))
                else:
                        row.append(dcell(x,y,(randint(0,max_range))))
```

```
        cells.append(row)

iterations = width+height+2

print "Simulating for ",iterations," iterations"

# iterate the cellular automata
for i in range(iterations):

        # save CA outputs as an image
        stri = ""
        if i < 1000: stri += "0"
        if i < 100: stri += "0"
        if i < 10: stri += "0"
        stri += str(i)
        i += 1
        outputImage2(prefix+stri+".png",cells)

        # make a copy of the present value of the cells
        for y in range(height):
                for x in range(width):
                        icells[y+1,x+1] = cells[y][x].val

        # use the copy to determine the next state of the cells (so update is synchronous
            )
        for y in range(height):
                for x in range(width):
                        cells[y][x].update()

        # test solution - how many cells match the required output?
        unmatched = 0
        for y in range(height):
                for x in range(width):
                        if cells[y][x].output != flag[y][x]:
                                unmatched += 1
        print "This configuration iteration matched all but ",unmatched," cells"
```

A.3 Self-assembling self-repairing ALU code

Listing A.5: VHDL of self-assembling self-repairing full-adder

```
library ieee;
use ieee.std_logic_1164.all;

-- a cell, one of 16 for each bit of ALU consists of:
        -- 2 state inputs, 2 state outputs
        -- 2 logic inputs, 2 logic outputs

entity cell is
        port (reset: in std_ulogic;
                a_in: in std_ulogic;
                b_in: in std_ulogic;
                c_out: out std_ulogic;
                d_out: out std_ulogic;
                n_in: in integer range 0 to 7; -- STATE input (North)
                w_in: in integer range 0 to 7; -- STATE input (West)
                e_out: out integer range 0 to 7; -- STATE output (East)
                s_out: out integer range 0 to 7); -- STATE output (South)
end entity cell;


architecture state_cell_assembler of cell is
        shared variable state : integer range 0 to 7;

                -- declare and populate the next-state look-up table
        type state_table is array (1 to 48,1 to 3) of integer range 0 to 7;
        constant state_lut : state_table := (

                                -- ADDER code (boundary conditions key: 1)
                (0,1,1),
                (0,4,1),
                (1,0,1),
                (1,2,3),
                (1,3,2),
                (1,7,3),
                (2,1,5),
                (3,2,4),
                (3,5,1),
                (4,0,1),
                (4,4,2),
                (5,1,7),
                (5,3,0),
                (7,3,4),

                                -- AND code (boundary conditions key: 2)
                (0,3,7),
                (0,5,3),
                (0,7,7),
                (0,2,7),
                (2,6,7),
                (2,7,5),
                (3,6,5),
                (4,5,3),
                (4,7,6),
                (5,0,2),
                (5,2,6),
                (5,5,7),
                (6,0,5),
                (6,7,5),
                (7,5,3),

                                -- SUBTRACT code (boundary conditions key: 4)
                (3,0,7),
                (4,3,2),
                (4,7,6),
                (5,4,7),
                (6,2,3),
                (7,0,4),
                (7,6,4),

                                -- OR code (boundary conditions key: 7)
                (0,0,3),
                (2,2,0),
                (3,3,3),
```

```
                (3,7,7),
                (7,2,0),
                (7,7,2),

                                -- NOT code (boundary conditions key: 5)
                (2,0,6),
                (0,6,4),
                (4,2,2),
                (4,6,7),
                (5,7,4),
                (7,4,6)
                                );

                -- determine the next-state of the cell
        begin
        determine_state: process(n_in,w_in,reset) is
                begin
                if reset = '1' then
                        state := 0;
                else
                        for i in 1 to 48 loop
                                if (state_lut(i,1) = n_in) and (state_lut(i,2) =
+w_in) then
                                        state := state_lut(i,3);
                                end if;
                        end loop;
                end if;
                s_out <= state;
                e_out <= state;
        end process determine_state;

                -- perform ALU function on logic inputs
        arithmetic: process(a_in,b_in) is
                variable ans : std_ulogic;
                begin
                        case state is
                                when 0 => ans := b_in;
                                when 1 => ans := a_in or b_in;
                                when 2 => ans := a_in;
                                when 3 => ans := b_in;
                                when 4 => ans := (a_in xor b_in) and a_in;
                                when 5 => ans := a_in xor b_in;
                                when 6 => ans := not a_in;
                                when 7 => ans := a_in;
                        end case;
                        c_out <= ans;
                        d_out <= ans;
        end process arithmetic;
end architecture state_cell_assembler;
```

A.4 Self-assembling 3D systems code

Listing A.6: Design of self-assembling 3D systems code

```
#!/usr/bin/python

import Image, ImageDraw, ImageFont, pygame
import operator
import time
from random import randint

# import the design as a 3D array
from flag import *

# save as
filename = 'transformer.jam'

depth = len(flag)
height = len(flag[0])
width = len(flag[0][0])

for z in range(depth):
        for y in range(height):
                for x in range(width):
                        if flag[z][y][x] != 0: flag[z][y][x] = 1

# Assigments store (output,aximuth)
assignments = {}
rules = {}
rulesc = {}

# store rule in an associative array
def makeRule(i,j,k,l,m):
        if not rules.has_key(i):
                rules[i] = {}
        if not rules[i].has_key(j):
                rules[i][j] = {}
        rules[i][j][k]=l

# create boundary cells in rcells array
bounds = []
from numpy import *
rcells = zeros((depth+2,height+2,width+2))
alive = 0
for z in range(depth):
        for y in range(height):
                for x in range(width):
                        rcells[z+1,y+1,x+1] = flag[z][y][x]
                        if flag[z][y][x] != 0: alive += 1

def n(zz,z,yy,y,xx,x):
        global bounds,cellsToSolve
        if (zz+z,y+yy,x+xx) in cellsToSolve: return int(rcells[zz+z+1,y+yy+1,x+xx+1])
        else: return 0

def n_(zz,z,yy,y,xx,x,p_arr):
        global bounds,cellsToSolve
        b = filter(lambda a: (z+zz,y+yy,x+xx) in a[1], p_arr)
        if (zz+z,y+yy,x+xx) not in cellsToSolve and len(b)!=0: return int(rcells[zz+z+1,y
            +yy+1,x+xx+1])
        else: return 0
def nn(zz,z,yy,y,xx,x):
```

```
        return int(rcells[zz+z+1,y+yy+1,x+xx+1])

def getInputs(z,y,x,azimuth):
        # Azimuth in range(1,8) - corresponds to corner of cube
        zf = 1; yf = 1; xf = 1;
        if azimuth in (1,2,3,4): zf = -1
        if azimuth in (3,4,7,8): yf = -1
        if azimuth in (1,3,5,7): xf = -1


        zn = n(zf*1,  z, yf*1, y, 0,    x)
        zw = n(zf*1,  z, 0,    y, xf*-1,x)
        zx = n(zf*1,  z, 0,    y, 0,    x)
        ze = n(zf*-1, z, 0,    y, xf*1, x)
        zs = n(zf*-1, z, yf*-1,y, 0,    x)
        ne = n(0,     z, yf*1, y, xf*1, x)
        ee = n(0,     z, 0,    y, xf*1, x)
        ss = n(0,     z, yf*-1,y, 0,    x)
        sw = n(0,     z, yf*-1,y, xf*-1,x)


        return (zn,zw,zx,ze,zs,ne,ee,ss,sw)

# responsible for determining azimuth of partition.  This is a recursive function.
ac = []
def detAzimuth((z,y,x)):
        ac.append((z,y,x))
        print "detecting azimuth for ",z,y,x,az
        global az,azimuth

        if az[0] == 0:
                if flag[z][y][x] == 0: az[0] = 1
                elif z > 0 and flag[z-1][y][x] not in (0,'X'): az[0] = 1
                elif z < (depth-1) and flag[z+1][y][x] not in (0,'X'): az[0] = 2
        if az[1] == 0:
                if flag[z][y][x] == 0: az[1]=1
                elif y > 0 and flag[z][y-1][x] not in (0,'X'): az[1] = 1
                elif y < (height - 1) and flag[z][y+1][x] not in (0,'X'): az[1] = 2

        if az[2] == 0:
                if flag[z][y][x] == 0: az[2]=1
                elif x > 0 and flag[z][y][x-1] not in (0,'X'): az[2] = 1
                elif x < (width - 1) and flag[z][y][x+1] not in (0,'X'): az[2] = 2

        if az[0] != 0 and az[1] != 0 and az[2] != 0:
                azmths =
                     [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
                azimuth = azmths.index(az) + 1
        elif (z-1,y,x) not in ac and az[0] != 0 and z > 0 and flag[z-1][y][x] not in (0,'
             X'): detAzimuth((z-1,y,x))
        elif (z+1,y,x) not in ac and az[0] != 0 and y > 0 and flag[z][y-1][x] not in (0,'
             X'): detAzimuth((z,y-1,x))
        elif (z,y-1,x) not in ac and az[1] != 0 and x > 0 and flag[z][y][x-1] not in (0,'
             X'): detAzimuth((z,y,x-1))
        elif (z,y+1,x) not in ac and az[1] != 0 and z < (depth-1) and flag[z+1][y][x] not
              in (0,'X'): detAzimuth((z+1,y,x))
        elif (z,y,x-1) not in ac and az[2] != 0 and y < (height-1) and flag[z][y+1][x]
             not in (0,'X'): detAzimuth((z,y+1,x))
        elif (z,y,x+1) not in ac and az[2] != 0 and x < (width-1) and flag[z][y][x+1] not
              in (0,'X'): detAzimuth((z,y,x+1))
        else:
                for i in range(3):
                        if az[i] == 0: az[i] = 1;
                azmths =
                     [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]

                print az
                azimuth = azmths.index(az) + 1

# responsible for dividing design into partitions.  This is a recursive function.
def explorer((z,y,x),azimuth):
        zf = 1; yf = 1; xf = 1;
        if azimuth in (1,2,3,4): zf = -1
        if azimuth in (1,2,5,6): yf = -1
        if azimuth in (1,3,5,7): xf = -1
        z2 = z+zf; y2 = y+yf; x2=x+xf;
        if z >= 0 and y >= 0 and x >= 0 and z < depth and y < height and x < width:
                try: a = cellsToSolve.index((z,y,x))
                except ValueError:
                        if flag[z][y][x] not in (0,'X'):
                                cellsToSolve.append((z,y,x))
                                flag[z][y][x] = 'X'

                                if (z-zf >= 0) and (z-zf < depth) and (flag[z-zf][y][x]
                                     not in (0,'X')):
                                        bounds.append((z-zf,y,x))
                                if (y-yf >= 0) and (y-yf < height) and (flag[z][y-yf][x]
                                     not in (0,'X')):
                                        bounds.append((z,y-yf,x))
                                if (x-xf >= 0) and (x-xf < width) and (flag[z][y][x-xf]
                                     not in (0,'X')):
                                        bounds.append((z,y,x-xf))
                                explorer((z2,y,x),azimuth)
                                explorer((z,y2,x),azimuth)
                                explorer((z,y,x2),azimuth)



# find first root
z = depth-1; y = height-1; x = 0;
while flag[z][y][x] == 0:
        x += 1
        if x == width: x = 0; y -= 1;
        if y == 0: x = 0; y = height-1; z-= 1;
bounds.append((z,y,x))

init_root = (z,y,x)

# explore design, partition and assign azimuths
partitions = []
root_inputs = []
while (1):
        try:
                root = bounds[0]
                bounds.remove(bounds[0])
        except IndexError:
                print "Design split into ",len(partitions),"partitions"
                break;

        azimuth = 0; az = [0,0,0]; ac=[];  detAzimuth(root)
```

```
        print "Exploring from",root,azimuth
        cellsToSolve = []; explorer(root,azimuth)
        bounds = [x for x in bounds if x not in cellsToSolve]
        print "cells in partition ,",len(cellsToSolve)
        partitions.append((root,cellsToSolve,azimuth))

p = 0

# solve for each partition
for (root,cellsToSolve,azimuth) in partitions:
        p += 1
        print "Solving for",len(cellsToSolve), "cells from",root,azimuth
        cellsToSolve.sort(lambda x,y: ((y[0]-root[0])**2 + (y[1]-root[1])**2 + (y[2]-root
             [2])**2  - (x[0]-root[0])**2 - (x[1]-root[1])**2 - (x[2]-root[2])**2))

        for (z,y,x) in cellsToSolve:
                rval = 0; tdir = []
                azmths =
                     [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]
                v=azmths[azimuth-1]
                zf = 1; yf = 1; xf = 1;
                if v[0] == 1: zf = -1;
                if v[1] == 1: yf = -1;
                if v[2] == 1: xf = -1;

                # transmission directions: 1,2,3,4 = n,e,s,w.  5,6 = up,down
                if(z,y,x) == (1,6,5): print "YO ",tdir
                # find tx dir for this azimuth
                if (z,y,x+xf) in cellsToSolve and (z,y,x+xf) != root:
                        if xf == 1: tdir.append(2)
                        else: tdir.append(4)
                if (z,y+yf,x) in cellsToSolve and (z,y+yf,x) != root:
                        if yf == 1: tdir.append(1)
                        else: tdir.append(3)
                if (z+zf,y,x) in cellsToSolve and (z+zf,y,x) != root:
                        if zf == 1: tdir.append(5)
                        else: tdir.append(6)


                if(z,y,x) == (1,6,5): print "YO ",tdir,p
                # find tx dir for new roots

                # is this cell an input to the root of another partition?
                sfr = (0,0,0)
                b = filter(lambda a: (z,y+1,x) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(1); sfr = (1,b[0][2])

                b = filter(lambda a: (z,y,x+1) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(2); sfr = (2,b[0][2])

                b = filter(lambda a: (z,y-1,x) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(3); sfr = (3,b[0][2])

                b = filter(lambda a: (z,y,x-1) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(4); sfr = (4,b[0][2])

                b = filter(lambda a: (z+1,y,x) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(5); sfr = (5,b[0][2])
                b = filter(lambda a: (z-1,y,x) == a[0], partitions[p:len(partitions)])
                if len(b) >= 1 and b[0][0] != root:
                        tdir.append(6); sfr = (6,b[0][2])

                if(z,y,x) == (1,6,5): print "YO ",tdir
                tdir.sort()
                tdir = set(tdir)
                if(z,y,x) == (1,6,5): print "YO ",tdir

                # solve next-state rule and assignments for this cell
                while(1):
                        solved = [1,2,3]; rval += 1

                        # is there a contradiction in the assignments array?
                        if rval in root_inputs: continue
                        try:
                                if assignments[rval] != (flag_orig[z][y][x],tdir):
                                        continue
                        except KeyError: pass

                        # is there a contradiciton in the rules array
                        try:
                                if rules[rval][zn][zw] != zx: continue;
                                else: solved[0] = 0
                        except: pass
                        try:
                                if rules[zs][rval][sw] != ss: continue;
                                else: solved[1] = 0
                        except: pass
                        try:
                                if rules[ze][ne][rval] != ee: continue;
                                else: solved[2] = 0
                        except: pass

                        # this cell is an input to the root of another partition, thus
                             check rules against zero-input combinations
                        if sfr[0] != 0:
                                try:
                                        a = rules[rval][0][0]
                                        continue
                                except: pass
                                try:
                                        a = rules[0][rval][0]
                                        continue
                                except: pass
                                try:
                                        a = rules[0][0][rval]
                                        continue
                                except: pass
                                root_inputs.append(rval)

                        # a solution has been found, save rules and assignments in array
                        for s in solved:
                                if s == 1:
                                                makeRule(rval,zn,zw,zx,(z,y,x))
                                elif s == 2:
                                                makeRule(zs,rval,sw,ss,(z,y,x))
```

```
                elif s == 3:
                                makeRule(ze,ne,rval,ee,(z,y,x))

                assignments[rval] = (flag_orig[z][y][x],tdir)
                rcells[z+1,y+1,x+1] = int(rval)
                rval = 0
                break

        # create the next-state rule for the root of this partition
        zf = -1; yf = -1; xf = -1;
        if azimuth in (1,2,3,4): zf = 1
        if azimuth in (1,2,5,6): yf = 1
        if azimuth in (1,3,5,7): xf = 1

        if root == init_root:
                makeRule(0,0,0,nn(0,root[0],0,root[1],0,root[2]),root)
        else:
                # what inputs are available:
                nnnn = n_(0,root[0],0,root[1],0,root[2],partitions[0:p])
                ee = n_(0,root[0],0,root[1],1,root[2],partitions[0:p])
                ss = n_(0,root[0],-1,root[1],0,root[2],partitions[0:p])
                ww = n_(0,root[0],0,root[1],-1,root[2],partitions[0:p])
                zx = n_(1,root[0],0,root[1],0,root[2],partitions[0:p])
                zzx = n_(-1,root[0],0,root[1],0,root[2],partitions[0:p])
                i1 = zzx; i2 = ss; i3 = ww;
                if zx != 0: i1 = zx
                if nnnn != 0: i2 = nnnn;
                if ee != 0: i3 = ee;
                makeRule(i1,i2,i3,nn(0,root[0],0,root[1],0,root[2]),root)

rules_count = 0;
for key,val in rules.items():
        for key2,val2 in val.items():
                rules_count += len(val2)

# save rules and assignments to file
f = open(filename,"w")
import pickle
pickle.dump((rules,assignments,depth,height,width,(init_root),flag_orig),f)
```

Listing A.7: Self-assemble and Render code for 3D systems

```
iterations = 100
interval = 3
import Image, ImageDraw, ImageFont, pygame
import operator
import time
from random import randint

root = (4,32,85)

import pickle

# load design file from
filename = ''

# cell class, responsible for getting inputs from neighbours, providing outputs to
    neighours, determining next-state and output.
class dcell:

        #initialize cell with location and state
        def __init__(self,x,y,z,init):
                self.x = x
                self.y = y
                self.z = z
                self.val = init
                self.next_val = init
                self.azimuth = []

                # determine output and azimuth from state
                try:
                        self.output = assignments[init][0]
                        self.azimuth = assignments[init][1]
                except:
                        self.output = 0
                        self.azimuth = []

        # from azimuth, determine appropriate inputs to next-state rule
        def getInputs(self):
                zn = 0; zs = 0; n = 0; s = 0; e = 0; w = 0; i = 0;
                if (self.z,self.y,self.x) in ic: i = 1;
                if 6 not in self.azimuth and self.z > 0: zn = int(cells[self.z-1][self.y
                    ][self.x].getOutput(5,i))
                if 5 not in self.azimuth and self.z < (depth-1): zs = int(cells[self.z
                    +1][self.y][self.x].getOutput(6,i))
                if 3 not in self.azimuth and self.y > 0: n = int(cells[self.z][self.y-1][
                    self.x].getOutput(1,i))
                if 1 not in self.azimuth and self.y < (height-1): s = int(cells[self.z][
                    self.y+1][self.x].getOutput(3,i))
                if 4 not in self.azimuth and self.x > 0: w = int(cells[self.z][self.y][
                    self.x-1].getOutput(2,i))
                if 2 not in self.azimuth and self.x < (width-1): e = int(cells[self.z][
                    self.y][self.x+1].getOutput(4,i))
                return ([zn,zs],[n,s],[e,w])

        # provide state-outputs to neighbours
        def getOutput(self,direction,i):
                if i == 1: print "Neighbour at ",(self.z,self.y,self.x)," settings: ",
                    direction,self.azimuth,self.val
                if direction in self.azimuth: return self.val
                else: return 0

        # determine next-state, output and azimuth from state-inputs
        def update(self):
                (zz,yy,xx) = self.getInputs()
                zz.sort(); yy.sort(); xx.sort();
                z = zz[1]; y = yy[1]; x = xx[1];

                try: self.next_val = int(rules[z][y][x])
                except KeyError: self.next_val = 0
                if (z,y,x) == (0,0,0) and (self.z,self.y,self.x) != root: self.next_val =
                    0
                if self.next_val == 0:
                        self.output = 0
                else:
                        try:
                                self.output = int(assignments[self.next_val][0])
```

```
                        self.azimuth = assignments[self.next_val][1]

                except KeyError:
                        self.output = 0
                        self.azimuth = []

# load rules and assignments from design file
init = 0
cells = []
f = open(filename,"r")
(rules,assignments,depth,height,width,root_temp,flag_orig) = pickle.load(f)

# initialize 3D world
for z in range(depth):
        slice = []
        for y in range(height):
                row = []
                for x in range(width):
                        if (z,y,x) == root: row.append(dcell(x,y,z,design_root))
                        elif init == 0: row.append(dcell(x,y,z,0))
                        else: row.append(dcell(x,y,z,(randint(0,max_range))))
                slice.append(row)
        cells.append(slice)

print "Simulating for ",iterations," iterations"

# enthought is the mayavi rendering libraries
import numpy
from enthought.mayavi.scripts import mayavi2
mayavi2.standalone(globals())
from enthought.tvtk.api import tvtk
from enthought.mayavi.sources.vtk_data_source import VTKDataSource
from enthought.mayavi.modules.outline import Outline
from enthought.mayavi.modules.surface import Surface
from enthought.mayavi.modules.iso_surface import IsoSurface

# for each iteration of design assembly
for i in range(iterations):

        # initialize root cell
        cells[root[0]][root[1]][root[2]].val = design_root
        cells[root[0]][root[1]][root[2]].next_val = design_root
        cells[root[0]][root[1]][root[2]].azimuth = [2,3,6]

        # update each cell synchronously
        for z in range(depth):
                for y in range(height):
                        for x in range(width):
                                cells[z][y][x].update()

for z in range(depth):
        for y in range(height):
                for x in range(width):
                        cells[z][y][x].val = cells[z][y][x].next_val

if (i % interval) == 0:
        # setup data to be rendered (from cell output)
        pppoints = []
        sppoints = []
        for z in range(depth-1,-1,-1):
                for y in range(height-1,-1,-1):
                        for x in range(width-1,-1,-1):
                                if cells[z][y][x].output != 0:
                                        pppoints.append([x,y,z])
                                        sppoints.append(cells[z][y][x].output)

        ff = numpy.zeros((depth+2,height+2,width+2))
        for z in range(depth):
                for y in range(height):
                        for x in range(width):
                                if cells[z][y][x].output !=0 :
                                        ff[z+1,height+1-y,x+1] = cells[z][y][x].
                                            output

        # render data using mayavi engine
        origin = numpy.array([0,0,0])
        dims = numpy.array([width+2,height+2,depth+2])
        spacing = numpy.array([1.0,1.0,1.0])
        mayavi.new_scene()
        sc = mayavi.engine.current_scene
        sc.scene.background = (1,1,1)
        spoints = tvtk.StructuredPoints(origin=origin, spacing=spacing,
            dimensions=dims)
        s = ff.transpose().copy()
        spoints.point_data.scalars = numpy.ravel(ff)
        spoints.point_data.scalars.name = 'flag'
        src = VTKDataSource(data = spoints)
        mayavi.add_source(src)
        iso = IsoSurface(compute_normals=True)
        iso.actor.property.opacity = 0.4
        mayavi.add_module(iso)
        sc.scene.reset_zoom()
```